VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KNIHOVNA PRO PRÁCI S TETRAEDRÁLNÍ SÍTÍ

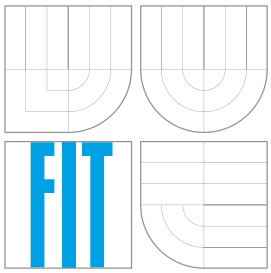
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DAVID HROMÁDKA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KNIHOVNA PRO PRÁCI S TETRAEDRÁLNÍ SÍTÍ

TETRAHEDRAL MESH PROCESSING LIBRARY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. DAVID HROMÁDKA
AUTHOR

VEDOUCÍ PRÁCE — Ing. MICHAL ŠPANĚL, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt

Mnoho inženýrských aplikací v architektuře, medicíně a strojírenství potřebuje vytvářet modely prostoru pro potřeby různých numerických výpočtů (např. FEM simulace). Tetraedrální sítě jsou jednou z perspektivních reprezentací těchto modelů. V této práci jsou popsány různé možnosti reprezentace tetraedrálních sítí vhodné pro jejich generování a zpracování. Je navržena knihovna pro zpracování sítě, která může být charakterizována úspornou reprezentací tetraedrální sítě se zachováním možnosti aplikovat na ni efektivně topologické a geometrické algoritmy. Knihovna je implementována v jazyce C++ s použitím šablon. Časová a prostorová složitost byla porovnána s knihovnou CGAL a podle výsledků měření má navržená knihovna nižší paměťové nároky než CGAL.

Abstract

Many architecure, medical and engineering applications need a spacial support for various numerical computations (i.e. FEM simulations). Tetrahedral meshes are one of perspective spatial representations for them. In this thesis, several possibilities of effective tetrahedral mesh representation for its generating and processing are described. A computer library for the mesh processing is proposed which can be characterized by memory efficient imposition of the mesh while preserving the ability to apply topological and geometric algorithms effectively. The library is implemented in C++ language using templates. Time and space complexity of typical mesh operations is compared with CGAL library and according to measurements the proposed library has lower memory requirements than CGAL.

Klíčová slova

Tetraedrální síť, C++, zpracování sítě, Delaunayho triangulace, zpracování, aplikace, topologie

Keywords

Tetrahedral mesh, C++, mesh processing, Delaunay triangulation, processing, application, topology

Citace

David Hromádka: Knihovna pro práci s tetraedrální sítí, diplomová práce, Brno, FIT VUT v Brně, 2013

Knihovna pro práci s tetraedrální sítí

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Ing. Michala Španěla, Ph.D.

.....
David Hromádka
15. května 2013

Poděkování

Děkuji Ing. Michalu Španělovi, Ph.D., za konzultace k této práci a za poskytnutí potřebné literatury.

Obsah

Definice a použité značení

Jako přirozená čísla jsou v této práci brána přirozená čísla včetně nuly.

- *Vertex* čili *vrchol* je základní element sítě. Jde o význačný bod v prostoru, který nesmí ležet uvnitř objemu ani uvnitř stěny žádného tetraedru sítě.
- *Hrana* čili *edge* je element sítě sestávající z dvouprvkové množiny vertexů.
- *Stěna* čili *face* je trojúhelník, který leží na hranici nějakého tetraedru v síti a každý jeho vrchol (vertex) je rovněž vrcholem daného tetraedru.
- *Halfface* je buď *vnitřní halfface*, která je dána stěnou (face) a tetraedrem, k němuž je příslušná; nebo *hraniční halfface*, která je dána stěnou a může ležet pouze na topologické hranici sítě.
- *Tetrahedron* čili *čtyřstěn* je element sítě daný čtyřprvkovou množinou vertexů.
- *Konektivita tetraedru* je čtyřprvková množina vertexů daného tetraedru.
- *Sousednost tetraedru* je čtyřprvková množina sousedních halffaces daného tetraedru, tedy všech halffaces v síti, které jsou dány stěnami tetraedru, ale nejsou mu příslušné (jde o halffaces sousedních tetraedrů, nebo hraniční halffaces).
- card(X) značí kardinalitu množiny X.
- Identifikátory volných funkcí jsou vyznačeny neproporcionálním písmem, např. takto: `volna_funkce()`.
- Identifiáktory členských funkcí tříd jsou uvedeny jako volné funkce, ale navíc předeslány tečkou: `.metoda_tridy()`.
- Šablonové funkce, má-li být zdůrazněna jejich šablonovitost, mají před kulatými závorkami uvedeny špičaté: `sablonova_funkce<>()`.

Kapitola 1

Úvod

Počítačové simulace prostorových veličin nalézají široké uplatnění v architektuře, medicíně, strojírenství a dalších aplikačních oblastech. Výpočty různých variant s cílem nalezení té optimální, nebo alespoň nějaké suboptimální, šetří peníze a stavební materiál a příležitostně zachraňují lidské životy. Mohlo by se zdát, že pro tyto účely stačí stávající postupy a algoritmy modelování, simulací a počítačové grafiky. Zatímco si ovšem syntéza obrazu pro účely zobrazení a animace téměř vždy vystačí s povrchovou reprezentací prostorových objektů, protože byly vyvinuty nesčetné technologie pomáhající povrchovost takových modelů zakrýt, mnoho ostatních, neméně důležitých, inženýrských aplikací, zejména simulace fluidní dynamiky, analýza medicínských dat či analýza zatížení strojních součástí metodou konečných prvků, vyžaduje plnohodnotné modelování prostoru a objektů v něm s možností řezů či nahlédnutí dovnitř těles.

Jednou z vhodných reprezentací je *tetraedrální síť*, tedy síť složená z obecných (tzn. ne nutně pravidelných) čtyřstěnů čili tetraedrů. Ve srovnání s volumetrickou reprezentací se tetraedrální síť jeví jako paměťově úsporná a efektivně využívající výpočetní výkon počítače. Aby však mohla být k výpočtům použita, musí být síť v počítači reprezentována vhodnou datovou strukturou.

Cílem této diplomové práce bylo navrhnout a implementovat knihovnu pro práci s tetraedrální sítí používající paměťově úspornou datovou strukturu založenou především na dynamických polích (v C++ `std::vector`). Výsledná knihovna nese název OTK (OpenTetraKernel) a poskytuje širokou škálu základních operací nad takto reprezentovanou tetraedrální sítí. Při návrhu knihovny byly v rámci možností zohledněny Delaunayho triangulace, které jsou zvláštním případem obecných tetraedrálních sítí a v inženýrských aplikacích jsou obzvlášť oblíbené pro svoje dobré vlastnosti[5].

Knihovna OTK je vhodná pro aplikaci v oblastech analýzy existujících sítí různými metodami, na příklad metodami prohledávání stavového prostoru jako BFS, DFS či backtracking. Velmi vhodná je pro experimenty v oblasti výtvarného počítačového umění, neboť nabízí volnost v používání netradičních geometrií. Také je vhodná pro simulace, v nichž nový stav buňky závisí jen na jejím blízkém okolí, či na generování tetraedrální sítě různými inkrementálními metodami.

Implementovaná knihovna nabízí funkce pro průchod sítí, označování tetraedru příznaky, efektivní ukládání uživatelských atributů k tetraedrům či vertexům (např. barvení) a vytváření a používání množin různých elementů sítě. Použití množinových operací umožňuje vyjádřit i složitější dotazy nad sítí, které by byly jinak obtížně realizovatelné. S množinami souvisí i možnost lokálních dotazů typu „zjistit množinu všech tetraedrů incidentních se zadaným vertexem“ apod.

V kapitole 2 jsou podrobně definovány základní pojmy související s vlastní tetraedrální sítí v geometrickém smyslu. Jsou tam definovány obecné tetraedrální sítě, triangulace a specificky Delaunayho triangulace.

V kapitole 3 jsou uvedeny různé dosud známé možnosti reprezentace tetraedrálních sítí v počítači a jsou analyzovány jejich výhody a nevýhody.

V kapitole 4 jsou představeny knihovny VectorEntity a CGAL, které se pro zpracování tetraedrálních sítí používají dnes.

V kapitole 5 je popsána architektura navržené knihovny a nároky kladené na její subsystémy. Jsou popsány použité datové struktury a vybrané algoritmy implementované v knihovně.

V kapitole 6 jsou popsány významné programovací techniky použité při vlastní implementaci knihovny v jazyce C++ (nesouvisející s tetraedrálními sítěmi).

V kapitole 7 jsou uvedena měření časové a prostorové složitosti vzorových aplikací používajících knihovnu OTK a je provedeno srovnání s knihovnou CGAL.

V kapitole 8 jsou navrženy vhodné aplikační oblasti pro knihovnu OTK a jsou uvedeny příklady vhodných rozšíření, na kterých by se mělo pracovat v případě dalšího vývoje knihovny.

Kapitola 2

Tetraedrální síť

Různé formy sítí (trojúhelníková, tetraedrální, hexahedrální apod.) jsou používány v širokém spektru inženýrských aplikací k modelování prostoru[5]. Vhodná metoda modelování prostoru je nezbytným předpokladem pro simulace jako např. metoda konečných prvků (FEM), které jsou potřeba zejména v architektuře a strojírenství.

Z tetraedrálních sítí jsou nejčastěji používány Delaunayho triangulace[5], protože jsou svými vlastnostmi pro simulace vhodnější než obecnější sítě; jsou manifold, konvexní a spojité a nevznikají v nich tetraedry s extrémně malými úhly, které by pak při simulaci způsobovaly výrazné numerické chyby.

V této kapitole popíšu obvyklou definici tetraedrální sítě podle [5]. Definice upravená pro potřeby knihovny OTK je uvedena v sekci 5.4.

2.1 Aplikace tetraedrální sítě

Tetraedrální sítě jsou používány v širokém spektru inženýrských oblastí, např v numerické matematice k aproximaci prostoru pro potřeby metody konečných prvků (FEM) [1]. Tato metoda je využívána ve výpočtech fluidní dynamiky (CFD), aerodynamiky, elektromagnetických polí, stavebnictví, chemickém inženýrství a při designu dopravních prostředků.

2.2 Tetrahedron

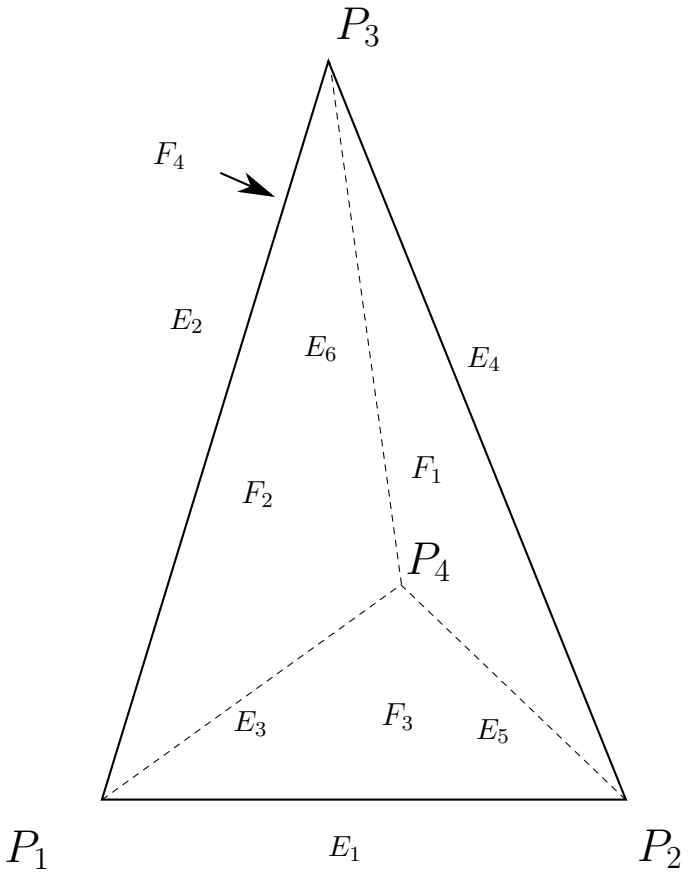
Tetrahedron čili čtyřstěn nebo také 3-simplex[5] je mnohostěn se čtyřmi stěnami. Může být plně definován čtveřicí vertexů (P_1, P_2, P_3, P_4). Existuje dvanáct možných permutací této čtveřice definujících orientovaný tetrahedron a 24 celkem.

Každá ze čtyř stěn tetraedru je dána uspořádanou trojicí vertexů:

- Stěna 1: (P_4, P_3, P_2),
- Stěna 2: (P_1, P_3, P_4),
- Stěna 3: (P_4, P_2, P_1),
- Stěna 4: (P_1, P_2, P_3).

Podobně může být šest hran tetraedru definováno implicitně jako dvojice vertexů:

- Hrana 1: (P_1, P_2),



Obrázek 2.1: Rozložení hran a stěn (faces) v tetraedru

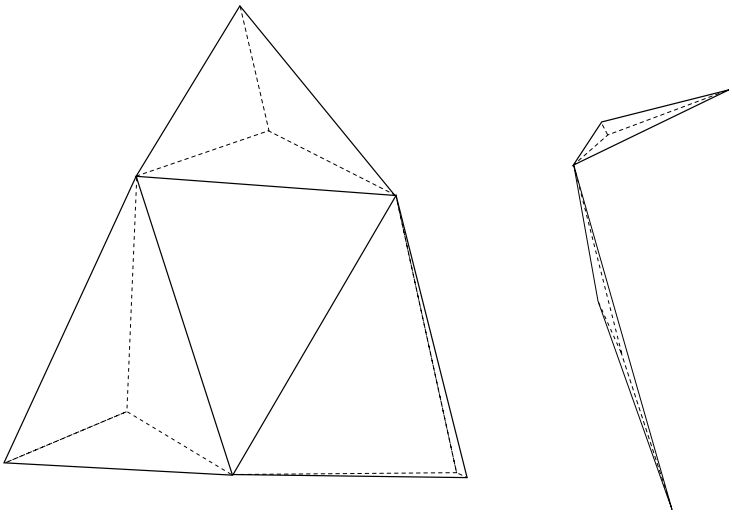
- Hrana 2: (P_1, P_3),
- Hrana 3: (P_1, P_4),
- Hrana 4: (P_2, P_3),
- Hrana 5: (P_2, P_4),
- Hrana 6: (P_3, P_4).

Toto uspořádání je ilustrováno na obrázku 2.1.

2.3 Obecná tetraedrální síť

Nechť Ω je uzavřená oblast v prostoru $\mathbb{R}^3$. Pak můžeme definovat tetraedrální síť τ jako množinu tetraedrů, která splňuje následující vlastnosti:

- Vlastnost 1: $\Omega = \bigcup_{T \in \tau} T$
- Vlastnost 2: Každý tetrahedron z τ je neprázdný.
- Vlastnost 3: Průnikem vnitřků tetraedrů $T_1, T_2 \in \tau$ takových, že $T_1 \neq T_2$, je prázdná množina.



Obrázek 2.2: Příklad obecné tetraedrální sítě

- Vlastnost 4: Průnikem tetraedrů $T_1, T_2 \in \tau$ takových, že $T_1 \neq T_2$, je prázdná množina, vertex, hrana, nebo trojúhelník.

Příklad obecné tetraedrální sítě je uveden na obrázku 2.2.

2.4 Delaunayho triangulace

V této sekci je definována Delaunayho triangulace.

2.4.1 Triangulace

Nechť τ je obecná tetraedrální síť podle výše uvedené definice a V je množina jejích vertexů. τ je *triangulace*, pokud splňuje tyto vlastnosti:

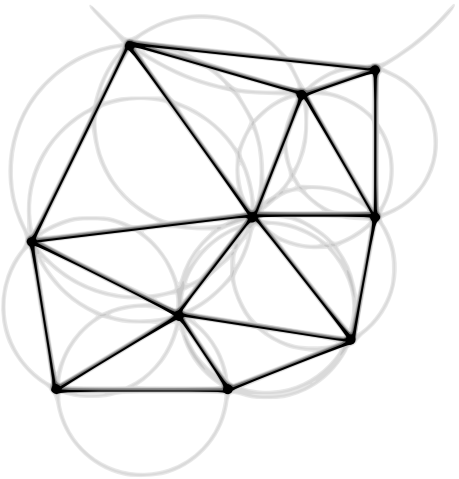
- Vlastnost 5: $\bigcup_{T \in \tau} T$ je konvexní obálka množiny bodů V.
- Vlastnost 6: $\forall T \in \tau : T = (P_1, P_2, P_3, P_4) \wedge P_1, P_2, P_3, P_4 \in V$.

2.4.2 Delaunayho vlastnost

Nechť τ je triangulace množiny vertexů V. τ je Delaunayho triangulace, pokud splňuje následující vlastnost:

- Vlastnost 7: Pro každý tetrahedron z τ platí, že jeho otevřená obalová koule neobsahuje žádný vertex z V.

Tato vlastnost se nazývá *kritérium prázdné koule*. Delaunayho triangulace je ilustrována na obrázku 2.3.



Obrázek 2.3: Delaunayho triangulace (obrázek převzat z [2])

Kapitola 3

Možnosti reprezentace tetraedrální sítě

Tetraedrální síť lze reprezentovat v počítači různými způsoby. Několik takových způsobů je představeno v [4].

3.1 Element connectivity

Klasickou reprezentací je *element connectivity*, tedy tabulka, kde každý řádek odpovídá jednomu tetraedru a obsahuje indexy čtyř jeho vertexů. Příklad této reprezentace je uveden v tabulce 3.1, odpovídající síť je vyobrazena na obrázku 3.1. Element connectivity je vhodnou reprezentací v případech, kdy nezáleží na pořadí zpracování tetraedrů. Sice úplně definuje topologii sítě, ale některé významné dotazy (nalezení sousedního tetraedru, test na hranici sítě) v ní nelze provést lokálně, takže časová náročnost zpracování rozsáhlejší sítě v této reprezentaci by neúměrně rostla.

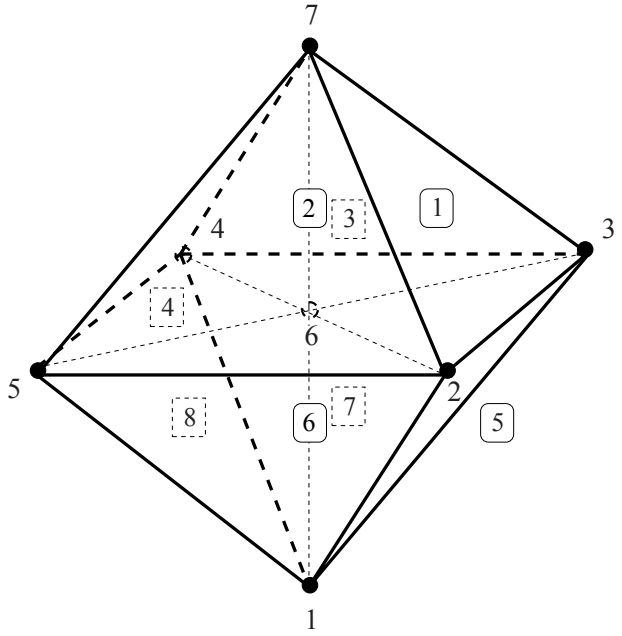
3.2 Array-based half-face data structure

V [4] je navržena datová struktura *half-face data structure* jako zobecnění dříve navržené datové struktury HEDS (Half-Edge Data Structure) pro povrchové sítě.

Pro dosažení kompaktnosti a úplnosti pro prostorové sítě, bylo navrženo zobecnění na polích založené HEDS zvané *half-face data structure* (HFDS). Jejich zobecnění je založeno

2	3	6	7
2	6	5	7
3	4	6	7
4	5	6	7
2	6	3	1
5	6	2	1
6	4	3	1
6	5	4	1

Tabulka 3.1: Příklad element connectivity (převzat z [4]).



Obrázek 3.1: Příklad tetraedrální sítě (převzat z [4]).

na zjištění, že stěny (faces) hrají v prostorových sítích podobnou roli jako hrany v povrchových sítích: Každá face je obsažena ve dvou buňkách (nebo v buňce a díře) a dvě kopie face mají opačné orientace, jsou-li jejich vertexy v každé buňce uspořádány podle pravidla pravé ruky (to znamená proti směru hodinových ručiček podle vnitřní normály face v buňce). Dvě kopie face nazývají half-faces a označují je jako navzájem *protější*. Analogicky k kódování half-edges v povrchových sítích, kódujeme každou half-face dvojicí čísel ⟨c,i⟩, kde c je ID obsahující buňky a i je index stěny (počínaje od 1) v buňce. Navíc přiřazujeme po sobě jdoucí ID hraničním faces (počínaje od 1) a kódujeme half-face s hraničním ID b jako ⟨b,0⟩. Protože je maximálně 6 stěn na buňku, můžeme zakódovat ID half-face v jednom integeru při použití nanejvýš tří bitů pro druhou část a zbytku bitů pro první část.

Výše uvedené kódovací schéma naznačuje zřejmé zobecnění HEDS se třemi poli: V2f, F2f a B2f, která jsou protějšky V2e, E2e a B2e, a ve kterém ID half-edges jsou nahrazena ID half-faces. Nicméně, není to ideální zobecnění, protože na rozdíl od E2e a V2e v HEDS, F2f a V2f již neposkytují kompletní datovou strukturu. Nekompletnost je způsobena faktem, že pořadí vertexů v half-face je cyklické bez záměrného počátečního vertexu, a tak je ne vždy možné dedukovat pořadí vertexů v buňce z části datové struktury. Je-li doplněno o element connectivity, může toto jednoduché zobecnění trpět neefektivitou, protože vyžaduje porovnání ID vertexů pro zarovnání half-faces při one-to-all dotazech na incidence.

K překonání těchto omezení definujeme *kotvu* half-face jako její záměrný první vertex a každá half-face pak má m ukotvených kopií, kde m je počet vertexů (nebo hran) face. Kódujeme *ukotvenou half-face* tříčástným ID ⟨c,i,j⟩, kde první dvě části korespondují s ID half-face a třetí koresponduje s *kotevním indexem* (počítaje 0), který je definován následovně: Pro nehraniční half-face, pokud kotva je k-tý vertex v seznamu vertexů face podle konvence CGNS, pak kotevní index je $k-1$; pro hraniční half-face, kotevní index je $\mathrm{mod}(m-t, m)$, kde t je kotevní index vertexu protilehlé half-face se stejnou kotvou. Dvě poslední části ID ukotvené half-face jsou tvořeny *místním* ID ukotvené half-face. Kotevní

index vyžaduje pouze dva bity a místní ID ukotvené half-face vyžaduje pouze pět bitů, takže plné ID ukotvené half-face může být kódováno jedním celým číslem. S 32-bitovými bezznaménkovými celými čísly je toto kódování dostatečné pro sítě obsahující až 2^{27} (tedy více než 100 milionů) buněk. Navíc přiřazujeme ID ukotvené half-face první hraně ukotvené half-face a pak obdržíme ID každé hrany v rámci každé face.

Podle konvence CGNS, každý vertex má místní index uvnitř buňky Je užitečné uložit mapování z místního ID ukotvené half-face na místní index vertexu uvnitř buňky. Přiřadíme jedinečné ID (mezi 1 a počtem typu mnohostěnů, obecně 4) každému typu elementu. K uložení mapování na typy elementů použijeme třídimenzní pole rozměru $4 \times 6 \times 4$, označené eA2v, jehož rozměry korespondují s typem ID, místních ID faces a kotevních indexů. Navíc definujeme pole eAdj stejné velikosti na uložení mapování z každé ukotvené half-face na místní ID její sousední ukotvené half-face uvnitř buňky kolem její první hrany. Nyní definujeme kompletní reprezentaci half-face data structrure:

- V2f: Mapuje každý vertex na ukotvenou half-face ukotvenou v daném vertexu; mapuje hraniční vertex na hraniční ukotvenou half-face.
- F2f: Mapuje každou nehraniční half-face s kotevním indexem 0 na její protilehlou half-face.
- B2f: Mapuje každou hraniční half-face s kotevním indexem 0 na její protilehlou half-face.

Analogicky k polím HEDS, V2f a B2f jsou hustá jednodimenzní pole, jejichž velikosti jsou rovny počtu vertexů, respektive hraničních faces. F2f je dvoudimenzní pole, kde každý řádek odpovídá buňce a počet sloupců je maximální počet faces v buňce, nabývající hodnoty mezi čtyřmi a šesti. Plná HFDS je tvořena těmito třemi poli doplněné o element connectivity. Konstrukce HFDS následuje proceduru podobnou té u HEDS s výjmkou dodatečných operací nutných k zarovnání protilehlých half-faces k určení kotevních indexů.

3.2.1 Shrnutí

Ve struktuře *array-based half-face data structure* [4] jsou tetraedry reprezentovány prostřednictvím tzv. AHF (anchored half-face). Každý tetrahedron v síti sestává z osmnácti AHF (šest stěn krát tři jejich vertexy). Identifikátor AHF se pak skládá ze tří složek:

- Indexu tetraedru.
- Indexu half-face v rámci tetraedru (0 až 3).
- Kotevního indexu (0 až 2) pro zajištění orientovanosti half-face k určitému vertexu tetraedru.

Vlastní datová struktura pak sestává ze tří polí:

- *V2f*, mapujícího každý vertex na nějakou AHF ukotvenou v daném vertexu (přednostně na hraniční AHF).
- *F2f*, mapujícího každou nehraniční AHF s kotevním indexem 0 na odpovídající protilehlou AHF.
- *B2f*, mapujícího každou hraniční AHF s kotevním indexem 0 na odpovídající protilehlou AHF.

Tato datová struktura umožňuje snadné procházení sítí a může být zkonstruována z element connectivity.

3.3 Další možnosti

Další možností jsou ukazatelové reprezentace, kde jsou odkazy na sousední tetraedry uloženy ve formě ukazatelů. Tyto reprezentace ovšem zejména na čtyřiašedesátibitových architekturách trpí velkou paměťovou náročností. Ty lze snížit nahrazením ukazatelů indexy do pole tetraedrů.

Kapitola 4

Existující řešení

Protože modelování prostoru pomocí tetraedrálních sítí je potřeba v architektuře, medicíně, strojírenství a dalších oborech lidské činnosti, byly již v minulosti různé knihovny na zpracování tetraedrálních sítí vyvinuty. V této kapitole jsou stručně nastíněny vlastnosti dvou vybraných z nich – knihoven VectorEntity a CGAL.

4.1 VectorEntity

VectorEntity je jednoduchá nízkoúrovňová knihovna implementovaná v jazyce C++. Jejím hlavním účelem je práce s vektorovou grafikou[10]. Je používána především pro zpracování medicínských dat, neboť je distribuována jako součást toolkitu MDSTk (Medical Data Segmentation Toolkit).

Knihovna je objektově orientovaná. Vertexy a z nich složené entity vyšší úrovně (hrany, trojúhelníky a tetraedry) jsou v paměti ukládány zvlášť a sousednost mezi entitami je uložena pomocí ukazatelů, což ovšem vede k vysokým paměťovým nárokům, zejména na čtyřiašedesátibitových architekturách procesorů. Proto tato knihovna není vhodná pro zpracování velkých sítí.

4.2 CGAL

Druhou často používanou knihovnou umožňující zpracování tetraedrální sítě je CGAL (Computational Geomery Algorithms Library). Jeho cílem je poskytnout robustní, obecné, efektivní, snadno použitelné a přizpůsobitelné algoritmy a datové struktury pro výpočetní geometrii [6].

Podstatnou částí knihovny je *jádro*, které určuje použité datové typy a metodu reprezentace souřadnic a rovněž poskytuje základní operace nad geometrickými primitivy, jako jsou např. body. Uživatel tak může volit z široké škály datových typů od výpočetně rychlého a paměťově úsporného typu `float` až po náročné typy poskytující exaktní aritmetiku[1].

CGAL nabízí dvě šablony jader – `CGAL::Cartesian<>` pro souřadnice vyjádřené v kartézské soustavě souřadnic, a `CGAL::Homogenous<>` pro homogenní souřadnice.

Některé algoritmy jsou implementovány jako volné funkce, jiné jsou implementovány jako objektově orientované třídy. Knihovnu CGAL lze tedy považovat za multiparadigmatickou.

[1] Exaktní aritmetikou se v knihovně CGAL rozumí taková aritmetika, na které lze založit geometricky přesné výsledky.

CGAL nabízí široké spektrum funkcí z oblasti výpočetní geometrie. Tématu této práce se týkají především následující funkce[2]:

- obecné triangulace (39),
- Delaunayho triangulace (39),
- regulární triangulace, známé také jako vážené Delaunayho triangulace[9] (39),
- datové struktury pro triangulace (40),
- periodické triangulace (41),
- generování 3D sítí (51).

[2]V závorce jsou uvedeny čísla příslušných kapitol v online uživatelské příručce knihovny CGAL.

Kapitola 5

Návrh a architektura knihovny pro zpracování tetraedrálních sítí

Cílem diplomové práce je navrhnout a implementovat v programovacím jazyce C++ knihovnu pro paměťově úsporné uložení tetraedrální sítě umožňující na tuto síť aplikovat lokální topologické a geometrické algoritmy s časovou složitostí nezávislou na celkovém rozměru sítě.

V této kapitole bude podrobně pojednán návrh knihovny včetně detailního obeznámení se s datovými strukturami a implementovanými algoritmy. Programátorské techniky použité při implementaci jsou popsány v kapitole 6 na straně 37.

Navržená a implementovaná knihovna nese název OTK, který je zkratkou z OpenTetra-Kernel.

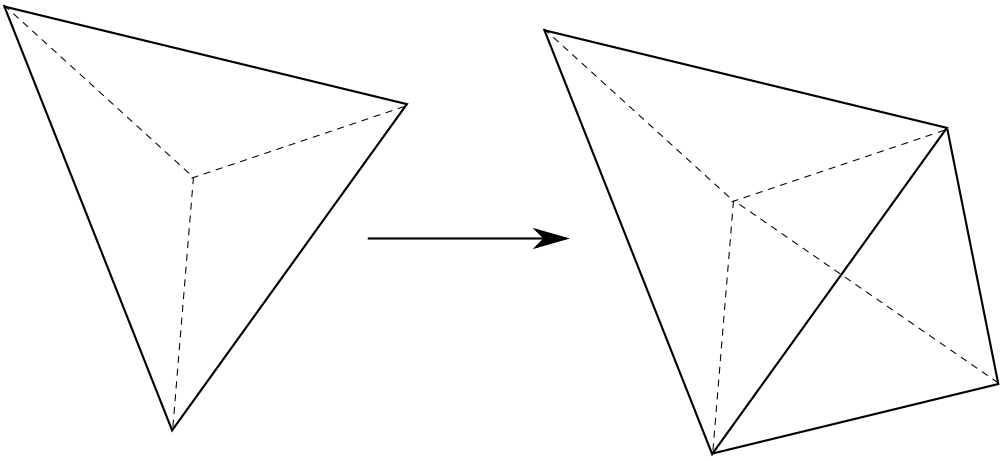
5.1 Obecné požadavky

- **co nejmenší paměťová náročnost** — Při zachování složitosti lokálních algoritmů nezávislé na velikosti sítě.
- **modularita** — Snadné napojení na další knihovny.
- **flexibilita** — Snadná konfigurace a přizpůsobení chování knihovny potřebám konkrétní aplikace.
- **přenositelnost** — Možnost používat knihovnu na různých zařízeních a různých operačních systémech. Nezávislost na hardwaru.
- **rozšířitelnost** — Možnost psát vlastní algoritmy pracující se sítí a používat je s původními algoritmy knihovny nebo místo nich.
- **schopnost pracovat s ne-manifold sítěmi** — Ale současně možnost zvolit rychlejší verze některých algoritmů, pokud je zaručeno, že zpracovávaná síť bude manifold.

5.2 Funkční požadavky

Knihovna poskytuje zejména následující operace (v závorce je uveden název příslušné funkce či datového typu):

1. Přidání jednotlivého tetraedru do sítě. (`add_tetrahedron()`) (Ukázka na obrázku 5.1.)
2. Přidání jednotlivého vertexu do sítě. (`add_vertex()`)
3. Smazání celé sítě/jen tetraedrů. (`clear_mesh()`) (Ukázka na obrázku 5.2.)
4. Vyhledání halfface podle trojice vertexů. (`find_halfface()`)
5. Vyhledání halfface podle hrany/dvojice vertexů. (`find_halfface_by_edge()`)
6. Vyhledání tetraedru podle čtveřice vertexů. (`find_tetrahedron()`)
7. Nastavení/testování příznaku *ke smazání* u tetraedru. (`get_delete_flag()`, `set_delete_flag()`).
8. Testování, zda byl vertex smazán. (`get_delete_flag()`).
9. Získání neplatného handle. (`get_invalid_handle()`)
10. Získání opačného halfface. (`get_opposite_halfface()`)
11. Práce s uživatelskými příznaky tetraedrů. (`flip_tetrahedron_flag()`, `get_tetrahedron_flag()`, `reset_tetrahedron_flag()`, `set_tetrahedron_flag()`)
12. Test, zda je element hraniční. (`is_border()`)
13. Test, zda je handle platné. (`is_valid()`)
14. Zjistění velikosti sítě/počtu elementů určitého typu. (`mesh_size()` a synonymní `n_bhfs()`, `n_tetras()`, `n_verts()`).
15. Odstranění osamělého vertexu nepoužitého v žádné závislé síti. (`remove_vertex()`) (Ukázka na obrázku 5.3.)
16. Rezervace paměti pro předpokládaný počet elementů. (`reserve()`)
17. Konverze mezi handle na vertex a jeho indexem v síti. (`handle_to_index()`, `index_to_handle()`)
18. Konverze mezi dvojicí vertexů a hranou. (`edge_to_vertices()`, `vertices_to_edge()`)
19. Další konverze mezi různými typy elementů: `tetrahedron_to_vertices()`, `tetrahedron_to_vertices_ordered()`, `tetrahedron_to_edges()`, `tetrahedron_to_halffaces()`, `tetrahedron_to_faces()`, `face_to_vertices()`, `face_to_edges()`, `face_to_halffaces()`, `face_to_tetrahedra()`, `halfface_to_vertices()`, `halfface_to_edges()`, `halfface_to_face()`, `halfface_to_tetrahedron()`.
20. Dotazy typu one-to-all: (`one_to_all<>()`).
21. Získání sousedních tetraedrů. (`get_neighbour_tetrahedron()`)

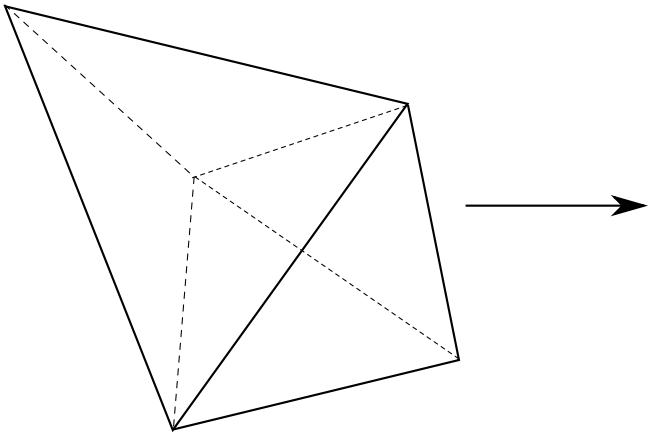


Obrázek 5.1: Přídání tetraedru do sítě (`add_tetrahedron()`).

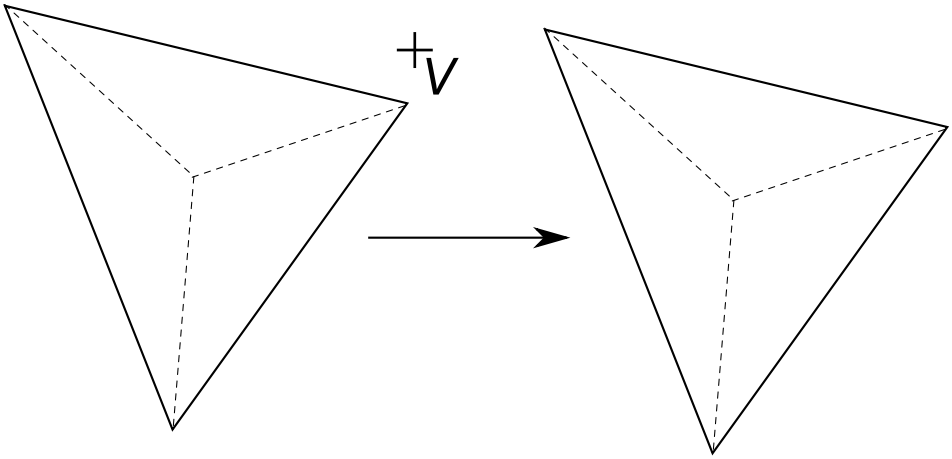
22. Odstranění tetraedrů označených ke smazání. (`garbage_collect()`)
23. Konstrukce sítě z element-connectivity s linearitmickou časovou složitostí. (`construct_mesh()`)
24. Vložení potenciální sítě do hlavní sítě. (`conjoin_mesh()`)

Knihovna dále poskytuje funkce pro práci s množinami elementů:

1. Procházení pomocí iterátorů. (`.begin()`, `.end()`)
2. Smazání množiny. (`.clear()`)
3. Zjištění velikosti/velikosti zařazené části. (`.size()`, `.commited_size()`)
4. Test prázdnosti množiny. (`.empty()`, `is_empty()`)
5. Vložení prvku/rozsahu zadaného iterátory. (`.insert()`, `insert_to()`)
6. Výměna vnitřního odkazu na data. (`.swap()`)
7. Zařazení dosud nezařazených prvků. (`commit()`)
8. Test přítomnosti prvku v množině. (`elem()`)
9. Test, zda jsou všechny prvky v množině zařazeny. (`is_commited()`)
10. Osamostatnění dat množiny od dalších odkazů. (`isolate()`)
11. Odstranění duplicit. (`unique()`)
12. Sjednocení/průnik/symetrická diference/rozdíl množin. (Pomocí operátorů `|`, `&`, `^`, `-`.)



Obrázek 5.2: Smazání celé sítě (`clear_mesh()`).



Obrázek 5.3: Smazání izolovaného bodu (`remove_vertex()`).

5.3 Omezení podoby tetraedrální sítě

Knihovna OTK klade na zpracovávanou síť následující omezení:

1. Každá tříprvková množina vertexů je sdílena 0, 1 nebo 2 tetraedry.
2. Každý tetrahedron je definován čtyřmi navzájem různými vertexy. Topologicky degenerované tetraedry jsou zakázány. Geometricky degenerované tetraedry mohou být reprezentovány pomocí několika vertexů s týmiž souřadnicemi.
3. Nesmí vadit, že každá face obsažená právě v jednom tetraedru je považována za hraniční.

5.4 Upravená definice tetraedrální sítě

V této sekci je uvedena definice tetraedrální sítě upravená pro potřeby návrhu knihovny OTK. Obvyklá definice tetraedrální sítě je uvedena v kapitole 2.

5.4.1 Neformální popis

Trojúhelník čili 2-simplex[5] je jedním z nejjednodušších geometrických útvarů v rovině. Má mnoho dobrých vlastností: na rozdíl od obecných polygonů je vždy konvexní a jeho tři body v prostoru vždy definují rovinu (výjimkou jsou degenerované trojúhelníky s nulovým obsahem). Proto jsou trojúhelníky oblíbeným základním stavebním kamenem většiny povrchových reprezentací těles v počítačové grafice[11].

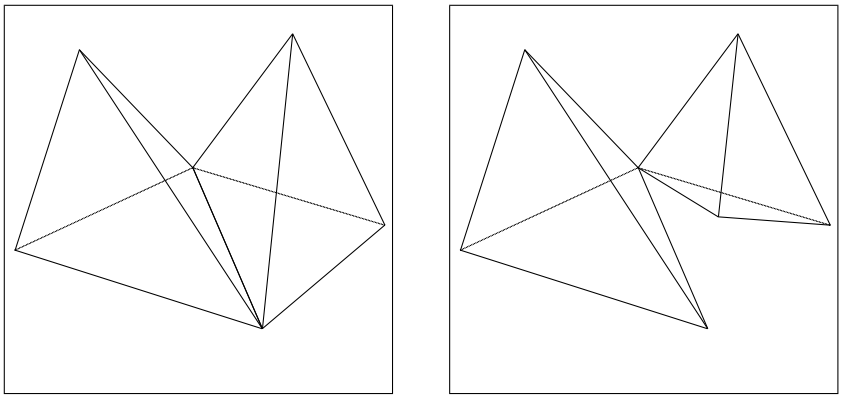
Trojúhelníků a jejich dobrých vlastností můžeme ale využít i pro objemovou reprezentaci těles. Sestavíme-li v prostoru ze čtyř trojúhelníků uzavřený útvar a vyplníme-li jeho vnitřní prostor, dostaneme tetrahedron (čtyřstěn) čili 3-simplex, základní stavební kámen tetraedrální sítě.

Tetraedrální síť je obdobou trojúhelníkové sítě (triangular mesh) popsané v [11]. Datová struktura popisující tuto síť bývá rozdělena do dvou logických částí – geometrické a topologické. Geometrická část přiřazuje vertexům jejich souřadnice, topologická část přiřazuje tetraedrům vertexy, z nichž sestávají, a může také obsahovat informace o sousednosti jednotlivých tetraedrů (tyto informace jsou redundantní, protože je lze odvodit od přiřazení vertexů tetraedrům).

Po tetraedrální síti požaduji, aby každou stěnu v síti sdílely nanejvýš dva tetraedry (jeden, pokud jde o hraniční stěnu; dva, pokud jde o vnitřní stěnu). Tetraedrální síť, která tuto podmínku nesplňuje, totiž v 3D prostoru nedefinuje těleso – některé tetraedry se překrývají nenulovým objemem – a ve 4D prostoru nedefinuje platný 3D povrch.

5.4.2 Manifold

Uvedená definice tetraedrální sítě je velmi abstraktní. Dovoluje popsat i objekty, které ve fyzikální realitě nelze vyrobit. Takové objekty se typicky vyznačují nekonečně tenkými hranami či bodovými spoji. Příklady takových objektů jsou uvedeny na obrázku 5.4. V praxi takové objekty obvykle není potřeba modelovat a správa datové struktury sítě dovolující takové objekty je náročnější než správa datové struktury, která je nedovoluje. Proto se definuje pojem *manifold* označující skupinu objektů, které by bylo možno reálně vyrobit. Tetraedrální síť modelující manifold musí splňovat navíc tyto dvě podmínky:



Obrázek 5.4: Ukázky non-manifold tetraedrálních sítí.

1. Každou hranu sdílejí nanejvýš dvě hraniční faces.
2. Pro každý vertex platí, že každé dvě s ním incidentní faces jsou vzájemně dosažitelné procházením po tetraedrech kolem daného vertexu.

5.4.3 Formální popis

Tetraedrální síť M definuji jako uspořádanou trojici $M = (V, g, T)$, kde:

- V je konečná množina vertexů,
- $g : V \to \mathbb{R}^D$ je geometrická funkce přiřazující každému vertexu číselné souřadnice v D-dimenzním lineárním prostoru,
- $D \geq 2$ je dimenze lineárního prostoru sítě
- a $T \subseteq \{u \mid u \subseteq V \wedge \text{card}(u) = 4\}$ je množina tetraedrů (konektivita sítě), která splňuje následující podmínku:

$$\forall x, y, z \in V : (\text{card}(\{x, y, z\}) = 3) \Longrightarrow (\text{card}(\{u \in T \mid \{x, y, z\} \subseteq u\}) \leq 2) \tag{5.1}$$

Pro definici tetraedrální sítě modelující manifold definuji nad každou tetraedrální sítí $M = (V, g, T)$ tyto pomocné funkce:

- Predikát bhf:

$$bhf(h) = (\text{card}(h) = 3 \wedge \text{card}(\{u \in T \mid h \subseteq u\}) = 1) \tag{5.2}$$

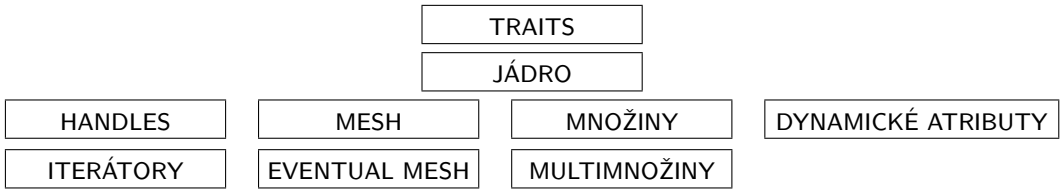
- Zobrazení $d : V \to (2^V)^2$:

$$\begin{aligned} d(v) = \{(h_1, h_2) \mid\ & (v \in h_1 \cap h_2)\ \wedge \\ & \wedge\ (\text{card}(h_1) = \text{card}(h_2) = 3)\ \wedge \\ & \wedge\ (\exists t_1, t_2 \in T : h_1 \subseteq t_1 \wedge h_2 \subseteq t_2)\} \end{aligned} \tag{5.3}$$

Tetraedrální síť modelující manifold musí navíc splňovat následující dvě podmínky (kde τ značí tranzitivní uzávěr relace):

$$\forall v_1, v_2 \in V : (v_1 \neq v_2) \Rightarrow (\text{card}(\{h \mid bhf(h) \wedge \{v_1, v_2\} \subseteq h\}) \leq 2) \tag{5.4}$$

$$\begin{aligned} \forall v, x_1, y_1, x_2, y_2 \in V : (\text{card}(\{x_1, y_1, v\}) = \text{card}(\{x_2, y_2, v\}) = 3) \Longrightarrow \\ \Longrightarrow (\{x_1, y_1, v\}, \{x_2, y_2, v\}) \in \tau\,(d(v)) \end{aligned} \tag{5.5}$$



Obrázek 5.5: Návrh architektury.

5.5 Architektura knihovny

Knihovna OpenTetraKernel se skládá z následujících subsystémů:

1. traits
2. kernel (jádro)
3. handles
4. iterators (iterátory)
5. mesh (síť)
6. eventual mesh (potenciální síť)
7. sets nad multisets (množiny a multimnožiny)
8. dynamic attributes (dynamické atributy)

Schéma architektury je uvedeno na obrázku 5.5.

5.5.1 Traits

Traits jsou třída parametrizující jádro, která umožňuje uživateli konfiguaci knihovny. Pomocí traits lze nastavit datové typy použité v datových strukturách knihovny a některé prvky chování algoritmů.

5.5.2 Jádro

Jádro knihovny OTK je soubor funkcí a datových typů parametrizovaný traits. Uživateli knihovny poskytuje pohodlný přístup k dalším datovým typům knihovny (handles, iterátorům, množinám apod.). Obsahuje také některé interní pomocné funkce. Jádro se v kódu typicky značí `K`. K nejdůležitějším datovým typům v jádře patří:

- `K::elem_tp` – Celočíselný typ používaný v celé knihovně k ukládání indexů a složených čísel.
- `K::elem_fast_tp` – Celočíselný typ používaný k výpočtům.
- `K::handle_tp` – Celočíselný typ používaný v handles (v této verzi synonymum k `K::elem_fast_tp`).

5.5.3 Handles

Každý element sítě má svoje jednoznačné celočíselné označení. Toto označení se nazývá *handle* a jeho prostřednictvím se k danému elementu přistupuje. Handles jsou implementována jako POD[1] obálky nad datovou složkou typu `K::handle_tp`. Ke každému druhu handle pak existuje zvláštní hodnota *neplatné handle*, kterou je možno získat pomocí funkce `OTK::get_invalid_handle()`. Handles jsou pak obdobou ukazatelů.

5.5.4 Iterátory

Iterátory umožňují nějakým způsobem procházet síť. Součástí knihovny OTK jsou především *jaderné iterátory* (kernel iterators). Jaderné iterátory jsou iterátory s náhodným přístupem, které procházejí vnitřní datovou strukturu, a tedy procházejí přes všechny elementy daného druhu včetně smazaných a neplatných. Knihovna poskytuje jaderné iterátory po vertexech, tetraedrech a hraničních halffaces.

Vedle toho by měly být součástí knihovny *přeskakující iterátory* (skipping iterators), což jsou obousměrné iterátory, které procházejí pouze platné elementy daného typu. Dosud však nebyly implementovány.

Kromě toho v knihovně existují ještě *iterátory po množinách a multimnožinách*, což jsou obousměrné iterátory procházející zařazené prvky množiny/multimnožiny elementů daného typu. Iterátory po množinách automaticky přeskakují duplicity.

5.5.5 Dynamické atributy

Dynamické atributy se dělí na husté, řídké a husté po podmnožinách.

- *Hustý atribut* je takový atribut, který nějaké hodnoty nabývá pro každý element daného typu (např. souřadnice vertexu). Husté dynamické atributy existují pro vertexy, tetraedry a hraniční halffaces.
- *Řídký atribut* je takový atribut, který pro většinu elementů daného typu žádnou hodnotu nemá, nebo má pro ně definovanou výchozí hodnotu. Řídké dynamické atributy by mohly existovat i pro další typy elementů (např. hrany), ale v této verzi knihovny nejsou implementovány.
- *Atribut hustý po podmnožinách* je takový, který nabývá hodnotu pro každý element daného typu, ale množství různých hodnot aktuálně se vyskytujících v síti je výrazně nižší než počet elementů daného druhu, a hodnoty se tedy často opakují. Takové atributy sice nejsou v knihovně přímo implementovány, ale je možno je doimplementovat pomocí překladové tabulky a uložení indexu do překladové tabulky jako hustého dynamického atributu.

V současné verzi jsou v knihovně implementovány pouze husté atributy.

5.5.6 Síť

Síť je datová struktura, ve které je uložena topologie sítě – konektivita a pomocné údaje o sousednosti tetraedrů. Datová struktura sítě je podrobně popsána v kapitole 5.6. V budoucích verzích knihovny se může tato datová struktura měnit, programátor by s ní proto měl

[1] plain old data

nakládat pomocí funkcí tvořících rozhraní knihovny namísto přímého přístupu k datovým složkám.

Souřadnice vertexů se ukládají jako dynamické atributy vertexů a knihovna OTK s nimi vnitřně nepracuje. Zvažoval jsem možnost uložit souřadnice a případné další atributy vertexů v poli V2HF spolu s označením přístupového halfface, ale zvětšení struktury záznamu by vedlo k častějším výpadkům stránky, protože algoritmy v praxi málokdy požadují jak přístupový halfface tak souřadnice. Většinou buď jednorázově požadují přístupový halfface (a pak prochází mezi sousedními tetraedry), nebo požadují souřadnice více vertexů (protože se souřadnicemi počítají). Z tohoto hlediska se jako lepší jeví kompaktnější struktura, kterých se do jedné paměťové stránky vejde víc, a tedy se sníží pravděpodobnost výpadku stránky při procházení.

5.5.7 Potenciální síť

Potenciální síť je obdoba hlavní sítě. Na rozdíl od ní je ovšem každá potenciální síť na některé hlavní síti závislá a sdílí s ní množinu vertexů. Do dané hlavní sítě pak může být vložena pomocí funkce `conjoin_mesh()`.

5.5.8 Množiny a multimnožiny

Množiny a multimnožiny elementů slouží k hromadnému zpracování elementů určitého typu a k dotazům nad sítí.

5.6 Datové struktury

5.6.1 Složená čísla

Většina nízkoúrovňových funkcí knihovny OTK pracuje s dvojicemi čísel (*základ*, *rozšíření*) zakódovaných do tzv. **složených čísel** následujícím způsobem:

- Základ (base, spodní část) je kódován ve spodních (`K::LOWER_BITS`) bitech, tedy v bitech číslo 0 až (`K::LOWER_BITS - 1`).
- Rozšíření (extension) je kódováno v bitech číslo (`K::LOWER_BITS`) až (`K::USE_BITS - 1`).
- Zbylé bity podkladového celočíselného typu jsou nevyužity a při práci s knihovnou jsou vždy nulové. To umožňuje použít jako podkladový i znaménkový celočíselný typ.

Každý **odkaz na halfface** je tvořen složeným číslem.

- Pokud je rozšíření odkazu rovno `OTK_BHFEXT`, jedná se o odkaz na hraniční halfface a základ odkazu značí index této halfface.
- Pokud je rozšíření odkazu rovno `OTK_HFEXT + X`, kde $X \in \{0, 1, 2, 3\}$, jde o odkaz na vnitřní halfface. Základ značí index tetraedru a X značí číslo halfface v rámci tetraedru (podle tabulky 5.3).

Odkaz na halfface má stejnou strukturu jako záznam vertexu typu A nebo B. De facto je záznam vertexu A či B odkazem na halfface.

Typ	Rozšíření	Hodnota	Význam základu
A	`OTK_BHFEXT`	0	Index hraniční halfface.
B	`OTK_HFEXT + X`	1 až 4	Index tetraedru.
C	`OTK_DELETED`	5	Index dalšího smazaného vertexu nebo `K::LBMASK`.
D	`OTK_SPECIAL_STATE`	6	0 ⇒vertex je izolovaný 1 ⇒vertex je non-manifold
E	`OTK_INVALID_STATE`	7	Nemá význam (neplatný záznam).

Tabulka 5.1: Typy záznamu vertexu

5.6.2 Záznam vertexu

Záznam vertexu se uvádí v datové struktuře sítě v tabulce V2HF a multimapě V2NHF. Existující typy záznamů vertexu jsou shrnuty v tabulce 5.1. Záznam vertexu představuje informaci o klasifikaci vertexu (hraniční, osamělý, manifold, non-manifold, smazaný, neplatný) a o přístupové halfface. Přístupová halfface je zvolená halfface z množiny halffaces vzájemně dostupných procházením po tetraedrech kolem daného vertexu.

Záznamy vertexu typu A a B představují odkazy na přístupové halffaces (hraniční a vnitřní). V multimapě V2NHF se smí vyskytovat záznamy pouze těchto typů.

Záznam vertexu typu C značí, že vertex byl smazán. Smazané vertexy jsou znovupoužívány při vkládání nových vertexů do sítě, pokud není toto chování zakázáno pomocí traits. Proto tvoří smazané vertexy zásobník, jehož dno je označeno zvláštní hodnotou `K::LBMASK`.

Záznam vertexu typu D značí buď osamělý vertex (základ 0), který může být smazán, pokud není využit v žádné závislé potenciální síti, nebo non-manifold vertex (základ 1), odkazy na jehož přístupové halffaces jsou uloženy v multimapě V2NHF.

Záznam vertexu typu E pak není používán a je neplatný.

Poznámky k přístupovým halffaces:

- Pokud je alespoň jedna ze vzájemně přístupných halffaces hraniční, je jako přístupová halffaces zvolena ta. (Díky tomu vertex, jehož žádná přístupová halfface není hraniční, nemůže být hraniční vertex.)
- Pokud má vertex víc přístupových halffaces, použije se záznam typu D se základem 1 a do multimapy V2NHF se k vertexu uloží všechny přístupové halffaces v záznamech typů A a B. I zde platí preference hraničních halffaces před vnitřními.
- Každé dvě přístupové halffaces k danému vertexu musí být procházením po tetraedrech kolem daného vertexu vzájemně nepřístupné. Redundantní halffaces jsou odstraňovány, jakmile je to možné.

5.6.3 Záznam tetraedru

Konektivita: Záznam tetraedru se skládá z konektivity a sousednosti. Konektivita obsahuje zejména čtveřici vertexů, kterou je tetrahedron definován. Tato čtveřice je v datové struktuře seřazena vzestupně, což umožnilo optimalizovat řadu algoritmů v knihovně. Indexy vertexů jsou uloženy jako základy složených čísel. Rozšíření tvoří dohromady 12 bitů použitých k uložení dalších informací:

- D... delete flag. Je-li nastaven na 1, je příslušný tetraedron označen ke smazání.
- P... číslo permutace použité k seřazení vertexů. Použitím inverzní permutace lze získat čtveřici vertexů tetraedru v původním pořadí. Hodnota se pohybuje v rozsahu 0 až 23, takže k jejímu zakódování stačí 5 bitů (`p_con[0]` obsahuje 2 horní bity a `p_con[1]` 3 spodní bity).
- U... uživatelské bitové příznaky. Knihovna musí nabídnout podle specifikace možnost uložit alespoň 4 takové příznaky ke každému tetraedru. Jejich výchozí hodnota je vždy 0. Implementovaná verze nabízí prostor pro 6 příznaků.
- C... indexy vertexů ve vzestupném pořadí. (Ve skutečnosti budou mít víc než 5 bitů.)

Způsob uložení konektivity v datové struktuře je ilustrován v tabulce 5.2.

Sousednost: Druhou částí záznamu tetraedru je sousednost, tedy odkazy na čtyři halffaces sousedící s jednotlivými halffaces tetraedru. Tyto odkazy jsou používány při procházení sítě a používá je rovněž funkce `OTK::get_opposite_halfface()`.

Při návrhu uspořádání halffaces jsem vzal v potaz skutečnost, že každá trojice ze čtyř vertexů tetraedru tvoří jednu z jeho halffaces. V každé halfface tedy chybí právě jeden z vertexů tetraedru a v každé jiný. Uspořádání jsem tedy navrhnul tak, že splňuje podmínku, že pro každé $X \in \{0, 1, 2, 3\}$ v halfface číslo X chybí vertex číslo $(3 - X)$. To umožnilo zjednodušit zápis některých algoritmů. Podrobně je uspořádání halffaces rozepsáno v tabulce 5.3.

5.6.4 Handles

Handles kromě edge handle jsou jednoduché POD obaly nad celočíselnou hodnotou typu `K::handle_tp`, na kterou je lze přetypovat. Proto se obvykle předávají hodnotou.

Edge handle je pak jednoduchá POD struktura s dvěma složkami typu `K::handle_tp`, pro kterou jsou definovány operátory porovnání, aby bylo možno ji použít stejně jako ostatní handles jako klíč v asociativních kontejnerech STL.

- Handle vertexu:

  Obsah platného handle: Index vertexu v rozsahu `0` až `K::MAX_INDEX`.
  Obsah neplatného handle: `K::LBMASK`

- Handle hrany (edge):

  Obsah platného handle: Indexy obou vertexů tvořících hranu v rozsahu `0` až `K::MAX_INDEX` tak, že `m_first` < `m_second`. „Smyčka“ tvořená jedním vertexem je zakázaná.
  Obsah neplatného handle: `K::LBMASK` v obou složkách.

p_con[0]	p_con[1]	p_con[2]	p_con[3]
DPPCCCCC	PPPCCCCC	UUUCCCCC	UUUCCCCC

Tabulka 5.2: Záznam konektivity tetraedru

Číslo halfface	Tvořena vertexy
0	$0, 1, 2$
1	$0, 1, 3$
2	$0, 2, 3$
3	$1, 2, 3$

Tabulka 5.3: Uspořádání halffaces v tetraedru

- Handle halfface:

  Obsah platného handle: Záznam vertexu typu A nebo B.
  Obsah neplatného handle: `K::INVALID_HANDLE`

- Handle face:

  Obsah platného handle: Záznam vertexu typu B.
  Obsah neplatného handle: `K::INVALID_HANDLE`

- Handle tetraedru:

  Obsah platného handle: Index tetraedru v rozsahu `0` až `K::MAX_INDEX`.
  Obsah neplatného handle: `K::LBMASK`

5.6.5 Tetraedrální síť (mesh)

V knihovně OTK se setkáme se dvěma typy sítě. Většina operací je definována pro obě tyto varianty. Jde o normální síť a potenciální čili závislou síť (eventual mesh). Datové struktury normální sítě jsou popsány zde. Datové struktury potenciální sítě jsou jim podobné; rozdíly jsou uvedeny v sekci 5.7.

Návrh datových struktur tetraedrální sítě je založen na [4]. Základem datové struktury jsou pole V2HF („vertex to halfface“), HF2HF („halfface to halfface“) a BHF2HF („border halfface to halfface“) a multimapa V2NHF (vertex to next halfface).

Pole V2HF mapuje jednotlivé vertexy na jejich tzv. *přístupové halffaces*, tedy takové halffaces, z nichž je možno procházením kolem vertexu dosáhnout všech s vertexem incidentních tetraedrů. Velikost tohoto pole přesně určuje počet vertexů v meshi včetně smazaných. Pro získání počtu nesmazaných vertexů je nutno odečíst zvlášť uložený počet smazaných vertexů (`m_free_vertices_count`).

Pokud síť nebude obsahovat žádné místo, kde by se dva tetraedry stýkaly vertexem aniž by se přitom stýkaly celou stěnou (face), bude pro každý vertex stačit jedna přístupová halfface. Knihovna OTK ovšem bude schopna pracovat i se sítěmi, které tuto podmínku splňovat nebudou. V takovém případě budou odkazy na přístupové halffaces,kterých bude v takovém případě víc, uloženy v multimapě V2NHF. Tato multimapa bude mapovat indexy vertexů na jejich přístupové halffaces.

Pole HF2HF obsahuje pro každý tetrahedron v síti jeho záznam sestávající z konektivity a sousednosti. V záznamu je rovněž uložen delete-flag, uživatelské příznaky a číslo permutace konektivity. Počet prvků pole HF2HF přesně odpovídá počtu tetraedrů v síti (včetně označených ke smazání).

Pole BHF2HF obsahuje pro každou hraniční halfface odkaz na její opačnou (opposite) halfface. Z definice hraničních halffaces to musí být reálně existující vnitřní halfface. Zaniklé

hraniční halffaces jsou vkládány do seznamu a znovupoužívány podle potřeby.

Datová struktura tetraedrální sítě rovněž obsahuje odkazy na všechny na ní závislé potenciální sítě a k ní patřící husté atributy.

Pole V2HF (`m_v2hf`) obsahuje pro každý vertex jeho záznam.

- Pokud existuje hraniční halfface incidentní s daným vertexem, ze které jsou procházením dosažitelné všechny tetraedry incidentní s daným vertexem, je zde uvedena.
- Jinak pokud existuje vnitřní halfface incidentní s daným vertexem, ze které jsou procházením dosažitelné všechny tetraedry incidentní s daným vertexem, je zde uvedena.
- Jinak je zde uvedeno složené číslo (1, `OTK_SPECIAL_STATE`), které značí, že vertex je non-manifold, a odpovídající záznamy jsou uloženy do multimapy V2NHF (`m_v2nhf`). Hraniční halffaces mají vždy přednost před vnitřními.

V důsledku těchto pravidel platí, že vertex je hraniční tehdy a jen tehdy, když existuje alespoň jedna z jeho přístupových halffaces, která je hraniční.

5.7 Potenciální síť (eventual mesh)

Potenciální síť se používá pro přípravu většího množství tetraedrů, které pak mohou být najednou vloženy do hlavní sítě, na které je potenciální síť závislá.

Datová struktura potenciální sítě se liší od datové struktury hlavní sítě v následujících bodech:

- Potenciální síť nemá multimapu V2NHF. Namísto toho je V2HF multimapou a plní současně úlohu V2HF a V2NHF. Potenciální síť sdílí množinu vertexů s hlavní sítí, na které je závislá. Proto není třeba počet vertexů v potenciální síti nikdy určovat.
- Na potenciální síti už nemůže být závislá další potenciální síť.

5.8 Množiny a multimnožiny

Množiny a multimnožiny elementů v knihovně OTK mají odkazovou sémantiku, což znamená, že jejich přiřazením se přiřazuje pouze odkaz na data a samotná data se nekopírují; jde tedy o mělkou kopii. Pokud programátor potřebuje vytvořit hlubokou kopii množiny, musí použít funkci `isolate()`.

Vnitřní datovou strukturou množin a multimnožin je tzv. *superblok* blíže popsaný v tabulce 5.4. Pole `m_data`, obsahující položky množiny, se dělí na dvě části – zařazené prvky a nezařazené prvky. Zařazené prvky jsou seřazeny vzestupně podle binárního komparačního funktoru `m_cmp`, a je tedy možné na ně aplikovat binární vyhledávání a další algoritmy vyžadující seřazenou posloupnost. Velikost části sestávající ze zařazených prvků je uložena

Superblok	
`long int m_refcount;`	`// počítadlo referencí`
`std::vector<T> m_data;`	`// položky množiny`
`size_t m_sorted_size;`	`// velikost zařazené části`
`std::less<T> m_cmp;`	`// porovnávací funktor`

Tabulka 5.4: Superblok množiny/multimnožiny

v datové složce `m_sorted_size`. Nezařazené prvky se nacházejí na konci vektoru a jsou uloženy v pořadí, v jakém byly do množiny vloženy. Prvky přidávané do množiny jsou vkládany na konec vektoru, kde jsou považovány za nezařazené až do volání funkce `commit()`, případně `commit_sorted()`. Většina funkcí množin a multimnožin pracuje pouze se zařazenými prvky.

Množiny a multimnožiny jsou zaměnitelné, vzájemně přiřaditelné a používají superblok téhož typu. Rozdíly mezi množinami a multimnožinami jsou následující:

- Iterátory množin automaticky přeskakují duplicity. (Pokud byly duplicity odstraněny např. funkcí `unique()`, může se vyplatit použít místo nich iterátory po multimnožinách, protože mohou být rychlejší.)
- Všechny funkce multimnožin kromě `unique()` zaručují zachování duplicit.

5.9 Funkce one_to_all

Funkce `one_to_all<>()` slouží k dotazům typu one-to-all nad sítí, tedy dotazům na lokální okolí určitého elementu. Funkce je ve skutečnosti šablona sdružující velké množství algoritmů a poskytující jim jednotné a přehledné rozhraní. Šablonové rozhraní funkce je tvořeno třemi parametry (čtvrtým parametrem je podtyp sítě a neměl by být explicitně uváděn):

1. `DST` – typ cílového elementu. Povinný parametr – musí být zadán.
2. `SRC` – typ zdrojového elementu. Bývá zadán, ale může být odvozen z typu předaného handle.
3. `K` – jádro. Bývá odvozeno od předané meshe.

Typy elementů se zadávají prostřednictvím k tomu určených výčtových konstant[2]:

- `OTK_VERTEX` – vertex.
- `OTK_EDGE` – hrana.
- `OTK_HALFFACE` – halfface.
- `OTK_TETRAHEDRON` – tetrahedron.

V této sekci uvedu přesnou specifikaci chování a účelu funkce `one_to_all<>()`, ale nejprve je pro účely tohoto popisu nutno definovat zobrazení v tak, aby splnilo následující čtyři podmínky:

1. $v(\texttt{X})$ je definováno pro každou výčtovou konstantu určenou k zadávání typu elementu.
2. Pokud $v(\texttt{X}) \leq v(\texttt{Y})$, pak pro každý uzavřený nedegenerovaný element typu `Y` označený y platí, že existuje element x typu `X` takový, že $x \subseteq y$.
3. Pokud $v(\texttt{X}) = v(\texttt{Y})$, pak $\texttt{X} = \texttt{Y}$.

[2] Existují ještě konstanty `OTK_FACE` a `OTK_BHFACE`, které ale jako parametr funkce `one_to_all<>()` nejsou přípustné.

Tyto podmínky splňuje na příklad zobrazení definované takto:

$$v(\texttt{OTK\_VERTEX}) = 1 \tag{5.6}$$
$$v(\texttt{OTK\_EDGE}) = 2 \tag{5.7}$$
$$v(\texttt{OTK\_FACE}) = 3 \tag{5.8}$$
$$v(\texttt{OTK\_TETRAHEDRON}) = 4 \tag{5.9}$$

Funkce `one_to_all<>()` je pak definována následujícím způsobem:

- `one_to_all<OTK_VERTEX, OTK_VERTEX>()` vrací množinu všech vertexů spojených se zadaným vertexem hranou existující v síti.
- `one_to_all<OTK_EDGE, OTK_EDGE>()` vrací množinu všech hran existujících v síti, jejichž průnikem se zadanou hranou je právě vertex.
- `one_to_all<OTK_HALFFACE, OTK_HALFFACE>()` vrací množinu všech halffaces, jejichž průnikem se zadanou halfface je právě hrana.
- `one_to_all<OTK_TETRAHEDRON, OTK_TETRAHEDRON>()` vrací množinu všech tetraedrů, jejichž průnikem se zadaným tetraedrem je právě trojúhelník.
- `one_to_all<X, Y>()`
  - Pokud $v(\texttt{X}) < v(\texttt{Y})$, vrací množinu takových elementů typu `X`, které jsou plně obsaženy v zadaném elementu typu `Y`.
  - Pokud $v(\texttt{X}) > v(\texttt{Y})$, vrací množinu takových elementů typu `X`, v nichž je plně obsažen zadaný element typu `Y`.

Úplný prototyp funkce je následující:

```
template<int A, int O, typename K, int V> inline
const typename K::set<A>::type one_to_all(
    const OTK::mesh<K, V> *p_mesh,
    OTK::uhandle<K, O> handle);
```

5.10 Algoritmy

5.10.1 Přidání jednotlivého tetraedru do sítě

Algoritmus: *Přidání jednotlivého tetraedru do sítě*

Vstup: Mesh, čtyři navzájem různé vertexy.
Výstup: Tatáž mesh rozšířená o nově vložený tetrahedron tvořený zadanými čtyřmi vertexy.

1. Vkládaný tetrahedron se dočasně přidá do HF2HF. Pokud algoritmus selže před potvrzením, bude záznam nového tetraedru odebrán.
2. Určit permutaci konektivity a vyplnit záznam o konektivitě tetraedru.
3. Zkontrolovat, zda jsou čtyři zadané vertexy různé.

4. Pokusit se najít ke každé halfface nově vznikajícího tetraedru ve stávající meshi všechny halffaces, které jsou tvořeny týmiž vertexy:
   - Pokud neexistuje žádná taková, bude třeba k nově vznikající halfface alokovat novou hraniční halfface.
   - Pokud existuje jedna vnitřní a jedna hraniční, bude třeba tu hraniční uvolnit a tu vnitřní připojit k nově vznikající halfface jako sousední.
   - Pokud existují dvě vnitřní, je třeba ohlásit chybu topologie sítě. V takovém případě zadaný tetrahedron nelze přidat.
5. Alokovat potřebné nové hraniční halffaces.
6. Alokovat husté atributy nového tetraedru.
7. Potvrdit přidání tetraedru. (Od tohoto kroku již nesmí dojít k výjimce.)
8. Původní hraniční halffaces určené k uvolnění nahradit v V2HF a V2NHF jejich sousedními vnitřními halffaces.
9. Uvolnit původní hraniční halffaces a připojit nový tetrahedron k jeho novým sousedům.
10. Pro všechny čtyři zadané vertexy vykonat korekci přístupových halffaces.

5.10.2 Algoritmus prvotního generování matice sousednosti

Vstupem pro knihovnu je tabulka *element connectivity*, která každému tetraedru v síti přiřazuje právě čtyři různé vrcholy z množiny vrcholů. Tato tabulka úplně definuje topologii tetraedrální sítě[4]. Z tabulky element connectivity je nutno zkonstruovat tabulky popsané v kapitole 5.6. V [4] se navrhuje použít k této konstrukci mapu (např. STL mapu). STL mapa má ale poměrně velkou spotřebu paměti. Standard C++ vyžaduje, aby byla zachována platnost iterátorů po přidání prvků do kontejneru[7]. To prakticky vylučuje implementovat mapu (map) či sadu (set) bez alokace alespoň dvou dodatečných ukazatelů na každou položku. Průchod následováním ukazatelů spojený s častými výpadky stránek pak výkon tohoto kontejneru dále snižují. Proto zvolený algoritmus místo toho používá STL vektor a provádí řazení sítě v rámci něj.

Algoritmus: *Prvotní generování matice sousednosti*

Vstup: Reprezentace tetraedrální sítě tabulkou *element connectivity*.
Výstup: Reprezentace tetraedrální sítě tabulkou *element connectivity* doplněná o tabulku sousednosti.

1. Alokovat STL vektor V a vyplnit jej handles na všechny nehraniční halffaces v síti.
2. Seředit vektor V podle trojic vertexů tvořících halffaces v lexikografickém pořadí.
3. Projít vektor V a kontrolovat počet sousedních halffaces sdílející stejnou trojici vertexů:

- Pokud je pouze jeden takový, jde o hraniční halfface. ⟶ Vygenerovat pro něj hraniční halfface.
- Pokud jsou dva takové, jde o sousední halffaces. ⟶ Vyznačit jejich sousednost do tabulky sousednosti.
- Pokud je víc takových, je síť neplatná. ⟶ Shodit výjimku.

5.10.3 Algoritmus garbage-collect

Tetraedry označené ke smazání nejsou ze sítě odstraňovány, ale pouze je u nich nastaven tzv. *delete-flag* (příznak smazání). Ten může být ještě zrušen. Tetrahedron označený ke smazání je v síti stále přítomen a blokuje přidávání konfliktních tetraedrů. K jeho skutečnému odstranění s případným vznikem díry v síti dojde až při volání příslušné funkce[3]. Přitom ztratí platnost handles na tetraedry a hraniční halffaces, a tato operace má navíc lineární asymptotickou složitost vzhledem k celkovému počtu tetraedrů v síti, takže je rozumné ji nepoužívat příliš často.

Kroky 5 a 6 v následujícím algoritmu nazývám *L-R metoda odstraňování tetraedrů*. Podstatou této metody je, že algoritmus postupuje střídavě zprava (od konce) a zleva (od začátku). Nejprve odstraňuje tetraedry zprava, dokud nenarazí na takový, který není označený ke smazání. Pak naopak vyhledává první tetrahedron zleva, který ke smazání označený je. Když se dostane do této situace, tetraedry na indexech L a R vymění a označený ke smazání (nově na indexu R) odstraní. L-R metoda končí, když $L > R$.

Algoritmus: *Operace garbage-collect*

Vstup: Tetraedrální síť.
Výstup: Tatáž síť, z níž byly odstraněny všechny tetraedry označené ke smazání.

1. Inicializovat počítadla:
   - $T_{td} = 0$ (tetraedry k odstranění),
   - $B_{tf} = 0$ (hraniční halffaces k uvolnění),
   - $B_{ta} = 0$ (hraniční halffaces k alokaci).
2. Pro každý tetrahedron v síti označený ke smazání:
   (a) Zvýšit počítadlo T_{td} o 1.
   (b) Pro každou jeho halfface, která sousedí s hraniční halfface, zvýšit počítadlo B_{tf} (danou hraniční halfface bude třeba zrušit).
   (c) Pro každou jeho halfface, která sousedí s tetraedrem, který *není* označený ke smazání, zvýšit počítadlo B_{ta} (po odstranění tohoto tetraedru bude nutno nahradit chybějící vazbu novou hraniční halfface).
3. Ošetřit okrajové podmínky:
   - Pokud nejsou žádné tetraedry ke smazání ($T_{td} = 0$), úspěšně skončit.

[3] `OTK::garbage_collect()`

- Pokud mají být smazány všechny tetraedry, zavolat funkci `OTK::clear_mesh()` a úspěšně skončit.

4. Rezervovat paměť pro B_{ta} nových hraničních halffaces. (Zbytek procesu nesmí být přerušen vznikem výjimek.)
5. Nastavit $L = 0$ a $R = T - 1$, kde T je počet všech tetraedrů v síti.
6. Dokud $L \leq R$:
   (a) Pokud je tetrahedron na indexu R označený ke smazání:
      i. Odpojit jej (algoritmus *odpojení tetraedru* na straně 33).
      ii. Snížit R o 1.
      iii. Jít zpět na krok 6.
   (b) Pokud je tetrahedron na indexu L označený ke smazání:
      i. Odpojit tetrahedron na indexu L (algoritmus *odpojení tetraedru* na straně 33).
      ii. Přečíslovat tetrahedron z indexu R na L.
      iii. Snížit R o 1.
   (c) Zvýšit L o 1.
7. Pro každou halfface v síti, která je v datové struktuře označena, že nemá platnou sousední halfface (toto označení vzniklo v algoritmu odpojování), alokovat novou hraniční halfface a zapsat ji do datové struktury sousednosti tetraedru.

Algoritmus: *Odpojení tetraedru*

Vstup: Tetraedrální síť (dat. struktura), handle konkrétního tetraedru v ní.
Výstup: Tatáž síť, v níž byly odkazy na daný tetrahedron označeny jako neplatné a byly uvolněny mu příslušné hraniční halffaces.

1. Pro každou halfface daného tetraedru:
   - Pokud sousedí s hraniční halfface, uvolnit danou hraniční halfface (použitím funkce jádra `K::free_bhf()`).
   - Pokud sousedí s halfface nějakého tetraedru, označit odkaz ze sousedního tetraedru v jeho datové struktuře sousednosti jako neplatný (implementováno přepsáním zvolenou konstantou).

5.10.4 Vložení potenciální sítě

Potenciální síť slouží k přípravě podsítě, která má být vložena do hlavní sítě po provedení operace garbage-collect. Potenciální síť je vždy vázána na jednu určitou hlavní síť, se kterou sdílí množinu vertexů. Operace vložení potenciální sítě do sítě hlavní je realizována následujícím algoritmem implementovaným ve funkci `OTK::conjoin_mesh()`.

Algoritmus: *Vložení potenciální sítě*

Vstup: Hlavní síť a na ní závislá potenciální síť.
Výstup: Hlavní síť rozšířená o tetraedry z potenciální sítě a nezměněná potenciální síť.

1. Pokud je potenciální síť prázdná, úspěšně skončit.
2. Alokovat dočasné pomocné pole `bhf_opps` o velikosti odpovídající počtu hraničních halffaces v potenciální síti. (V tomto poli se shromáždí odkazy na hraniční halffaces v hlavní síti, které odpovídají hraničním halffaces potenciální sítě.)
3. Nastavit $B_{ta} = 0$ (počet hraničních halffaces k alokování).
4. Najít ke každé hraniční halfface v potenciální síti odpovídající halfface v hlavní síti pomocí funkce `OTK::find_halfface()`.
   - Pokud nebyla žádná halfface nalezena, zapsat do pole `bhf_opps` na příslušný index neplatné handle a zvýšit B_{ta} o 1.
   - Pokud byla nalezena hraniční halfface, zapsat odkaz na ni do `bhf_opps`.
   - Pokud byla nalezena dvojice sousedních halffaces uvnitř sítě, skončit s chybou. (Další halfface nelze vložit, protože by to vedlo k topologické nekonzistenci sítě.)
5. Rezervovat paměť pro B_{ta} nových hraničních halffaces v hlavní síti.
6. Vložit všechny tetraedry z potenciální sítě na konec datové struktury hlavní sítě. Přitom:
   - Provést přečíslování tetraedrů. (Tetraedry jsou vkládány v pořadí, takže stačí ke každému indexu přičíst původní počet tetraedrů v hlavní síti.)
7. Projít nové tetraedry a popřipojovat jejich hraniční halffaces podle pole `bhf_opps`. Místo neplatných handles alokovat nové hraniční halffaces.
8. Uvolnit všechny hraniční halffaces, na které jsou odkazy v poli `bhf_opps`.
9. Pro každý záznam v tabulce V2HF potenciální sítě zkontrolovat, zda je daná halfface přístupná podle V2HF (resp. V2NHF) hlavní sítě, a pokud ne, přidat ji tam.

5.10.5 Zařazování prvků množin (commit)

Jak je popsáno v sekci 5.8, prvky přidávané do množin (resp. multimnožin) elementů jsou vkládány na konec pole `m_data`, kde zůstanou, dokud nebudou zařazeny funkcí `commit()`.

Funkce `commit()` implementuje následující algoritmus:

Algoritmus: *Zařazování prvků množin*

Vstup: Množina/multimnožina elementů.
Výstup: Tatáž množina/multimnožina elementů, v níž jsou ovšem všechny prvky zařazeny.

1. Pokud jsou všechny prvky zařazeny, úspěšně skončit.
2. Seřadit dosud nezařazené prvky[4].
3. Sloučit pole zařazených a nezařazených prvků (pomocí algoritmu standardní knihovny šablon `std::inplace_merge()`).
4. Nastavit počet zařazených prvků na celkový počet prvků v množině/multimnožině.

5.11 Jádro jako rozhraní

Jádro knihovny OTK poskytuje uživateli pohodlné rozhraní k datovým typům knihovny OTK. V následujícím přehledu jsou uvedeny pro srovnání syntaxe používající rozhraní zprostředkované jádrem a syntaxe přistupující k datovým typům OTK přímo. Syntaxi druhého typu je nutno použít v šablonovém kódu, má-li fungovat automatické odvození jádra od předaného formálního parametru.

V následujícím přehledu je uvedeno srovnání obou rozhraní (rozhraní jádra a základního rozhraní). Vysvětlivky k uvedeným zástupným identifikátorům jsou uvedeny v tabulce 5.5.

Handles

Rozhraní jádra ..... `K::handle<EL>::type`
Základní rozhraní .. `OTK::uhandle<K, EL>`

Iterátory

Rozhraní jádra ..... `K::iterator<IT_TYPE | EL>::type`
Základní rozhraní .. `OTK::kiterator<K, EL>`

Atributy

Rozhraní jádra ..... `K::attribute<DS | EL, T>::type`
Základní rozhraní .. `OTK::dense_attribute<K, EL, T>`

Množiny

Rozhraní jádra ..... `K::set<ELEMENT>::type`
Základní rozhraní .. `OTK::otk_set<K, EL, 0>`

[4] `std::sort()`

Vysvětlivky:	
K	Jádro.
EL	Typ elementu. Jedno z `OTK_VERTEX`, `OTK_EDGE`, `OTK_HALFFACE`, `OTK_FACE`, `OTK_TETRAHEDRON`, `OTK_BHFACE`.
IT_TYPE	Typ iterátoru (v této verzi jen `OTK_KERNEL`).
DS	Typ atributu (v této verzi jen `OTK_ATTR_DENSE`).
HF	Číslo halfface (`0`, `1`, `2`, nebo `3`).

Tabulka 5.5: Vysvětlivky k přehledu v sekci 5.11.

Multimnožiny

Rozhraní jádra ..... `K::multiset<EL>::type`
Základní rozhraní .. `OTK::otk_set<K, EL, 1>`

Zástupci halffaces

Jediné rozhraní..... `K::halfface<HF, false>(P_TH_CON)`

Neplatné handle

Rozhraní jádra ..... `K::get_invalid_handle()`
Základní rozhraní .. `OTK::get_invalid_handle<K>()`

Kapitola 6

Implementace

V této kapitole budou uvedeny některé programovací techniky použité při vlastní implementaci knihovny OTK v jazyce C++. Implementační detaily (jako např. názvy funkcí) jsou pro přehlednost uvedeny u návrhu knihovny v kapitole 5.

Knihovna OTK je implementována v programovacím jazyce C++ s využitím standardní knihovny C++ včetně STL. Dále využívá malou část mé pomocné knihovny DH_kernel, která také vyžaduje jen standardní knihovnu C++ podle standardu z roku 1998 doplněnou o čtyřiašedesátibitové celočíselné datové typy. Díky tomu může být knihovna OTK snadno přenesena na každou platformu, pro kterou existuje překladač C++ splňující standard ISO C++ 1998 a poskytující 64-bitové celočíselné typy.

6.1 Šablonové metaprogramování

V knihovně OTK byly využity některé techniky šablonového metaprogramování. Na příklad šablona `im::v_in_hf<>` simuluje 2D pole představující mapování vertexů jednotlivých v tetraedru na vertexy tetraedru. Dále knihovna obsahuje šablony `enum_type_to_value<>` a `enum_value_to_type<>`, poskytující mapování vybraných výčtových konstant na jejich typy a zpět.

6.2 Königovo vyhledávání

Většina datových typů OTK (mesh, handles, iterátory, ...) je definována ve jmenném prostoru OTK, kde je rovněž definována většina funkcí s nimi pracujících. Díky Königovu vyhledávání proto není při volání těchto funkcí nutné explicitně uvádět jmenný prostor OTK.

Na výčtové konstanty se Königovo vyhledávání nevztahuje. Knihovna proto poskytuje vnořený jmenný prostor `OTK::enums`, který obsahuje pouze vybrané výčtové konstanty s prefixem `OTK_`.

6.3 Univerzální rozhraní jádra

Jádro (šablona `OTK::kernel<>`) poskytuje jednotný přístup k handles, iterátorům, atributům a dalším datovým typům knihovny. Toto rozhraní je blíže popsáno v sekci 5.11.

Pro programátora je podstatné nezapomenout na `::type` na konci každé specifikace, součástí jádra je totiž pouze rozhraní vytvořené prostředky šablonového metaprogramování.

Vlastní datové typy jsou zpravidla definovány na úrovni jmenného prostoru OTK jako šablony parametrizovány jádrem.

Někdy programátor potřebuje napsat obecný kód fungující s jakýmkoliv jádrem. V takovém případě by mohl napsat např.:

```
template<typename K>
int commit_it(typename K::evmesh *p_mesh);
```

Tato deklarace ovšem nebude fungovat bez explicitní specifikace jádra při volání funkce, protože kompilátor si nemůže odvodit typ jádra z typu formálních parametrů. Řešením je použít složitější deklaraci založenou na skutečných datových typech OTK namísto rozhraní poskytovaného jádrem:

```
template<typename K, int V>
int commit_it(OTK::mesh<K, V> *p_mesh);
```

Většina funkcí knihovny OTK je deklarovaná tímto způsobem.

6.4 Získání neplatného handle

Funkce `OTK::get_invalid_handle()` je implementována ve třech variantách s různým počtem šablonových parametrů:

- Varianta bez parametrů vrací objekt s šablonovým operátorem přetypování, který umožňuje přetypovat jej na jakýkoliv typ handle podle jakéhokoliv jádra.
- Funkce explicitně parametrizovaná jádrem vrací objekt, který může být přetypován na kterýkoliv typ handle podle zadaného jádra.
- Funkce parametrizovaná jádrem a typem elementu vrací handle požadovaného typu bez nutnosti přetypování. Tato varianta je výhodná v případě, kdy kompilátor nemůže z kontextu odvodit požadovaný typ handle.

6.5 Paradigma knihy a zdrojové soubory

K organizaci zdrojových souborů knihovny jsem použil lehce upravené *paradigma knihy* podle [8] (str. 809 - 810). Podstatou tohoto paradigmatu je podle [8] uvažovat o kódu jako o speciální formě knihy a formátovat jej obdobným způsobem. Tím lze dosáhnout podstatného zvýšení přehlednosti kódu a při jeho čtení pak podobných výsledků jako při čtení knih.

Na počátku kódu se nachází „předmluva“, tj. úvodní komentář se stručným popisem významu souboru.

Následuje „obsah“ představující strukturu kapitol a podkapitol v souboru. Každá kapitola i podkapitola má své označení v hranatých závorkách, podle něhož ji lze v souboru snadno najít. „Obsahy“ souborů byly automaticky generovány jednoduchou pomocnou utilitou.

Tělo souboru je pak rozděleno do číslovaných kapitol a podkapitol. Jejich hlavičky mají pevně dané formátování, které umožňuje rychlou vizuální orientaci.

V [8] se mluví ještě o křížových odkazech, ale ty v kódu OTK nepoužívám.

V knihovně OTK se toto uspořádání osvědčilo, po čase se však ukázalo vhodné kód rozdělit do několika samostatných souborů:

- `otk.h` – Základní kód knihovny.
- `otk_exp_i.h` – Funkce tvořící vnější rozhraní knihovny.
- `otk_int_i.h` – Některé vnitřní pomocné funkce.
- `otk_permute_i.h` – Kód pro práci s permutacemi čtyřprvkových množin.
- `otk_set.h` – Kód množin a multimnožin.

Uživatel by měl vkládat pouze `otk.h` a `otk_set.h`. Ostatní hlavičkové soubory slouží pouze pro vnitřní rozdělení knihovny.

Ve zdrojových kódech je zavedena tříúrovňová hierarchie. Nejvyšší úroveň představují zdrojové soubory, každý z nich se dělí na kapitoly a ty na podkapitoly. Jednotlivé podkapitoly nejčastěji obsahují jen jednu funkci, v některých případech ovšem jedna kapitola zahrnuje i více souvisejících funkcí, zejména pokud jsou stejného jména.

Kapitola 7

Příklady, experimenty, srovnání

Jedním z cílů této diplomové práce bylo porovnat implementovanou knihovnu s jinou knihovnou používanou v téže aplikační oblasti, tedy v oblasti zpracování tetraedrálních sítí. Pro toto srovnání jsem si zvolil knihovnu CGAL. Navrhnul a implementoval jsem tři ukázkové příklady. Každý z nich jsem implementoval ve dvou verzích – jedné s použitím knihovny CGAL a druhé s použitím knihovny OTK. Použil jsem rovněž knihovnu Eigen.

Klíčovými vlastnostmi knihoven, které je třeba porovnat, jsou časová a prostorová složitost.

Časová složitost je množství výpočetního času procesoru, které daná aplikace vyžaduje pro svůj běh. Při návrhu knihovny jsem se snažil o zachování přijatelných hodnot tohoto parametru, protože příliš velká časová složitost může být příčinou praktické nepoužitelnosti knihovny. Uživatelé nemohou čekat na výsledek výpočtu desítky let.

Prostorová složitost je množství paměti, které aplikace pro svůj běh využije. Velká prostorová složitost aplikací využívajících stávající knihovny pro zpracování tetraedrálních sítí byla jednou z příčin vzniku knihovny OTK. V [4] byla navržena paměťově úsporná datová struktura *array-based half-face data structure*, popsaná v sekci 3.2. Knihovna OTK implementuje upravenou podobu této datové struktury, popsanou zejména v sekci 5.6, měla by tedy vykazovat nižší paměťovou náročnost než CGAL.

Protože se mi nepodařilo najít žádnou implementaci měřícího systému, která by vyhovovala stanoveným požadavkům na měření, navhnul a implementoval jsem vlastní metody měření časové a prostorové složitosti, popsané v sekcích 7.1 a 7.2.

7.1 Měření časové složitosti

K měření časové složitosti jsem implementoval objektově orientovanou třídu `stopky`, která funguje podobně jako skutečné stopky. Vlastní měření proběhla na operačním systému Ubuntu (Linux).

Aby bylo měření časové složitosti dostatečně vypovídající, před zahájením každého měření jsem měřenému procesu nastavil nejvyšší možnou prioritu.

Zvažoval jsem možnost použití linuxové utility `time`, ale ta měří vždy dobu běhu procesu jako celku; neumožňuje měřit dobu běhu jednotlivých částí kódu.

7.2 Měření prostorové složitosti

Množství spotřebované paměti měřím opakovaným voláním linuxového příkazu `free` v intervalu 0,5 sekundy. Tento příkaz vrací údaje o množství volné a použité paměti v systému[3]. Porovnáním velikosti obsazené paměti před a po spuštění procesu lze spolehlivě zjistit skutečné množství paměti spotřebované procesem.

Původně jsem chtěl množství alokované paměti měřit pomocí nástroje `valgrind`. Narazil jsem ovšem na problém, protože `valgrind` nahrazuje standardní alokační funkce svými, které ovšem zřejmě vrací ukazatele, které nemusejí být zarovnány na 16 bajtů, což způsobuje pád programu při provádění kódu, který na toto zarovnání spoléhá.

Pokoušel jsem se rovněž sledovat spotřebu paměti konkrétního procesu pomocí příkazů `top` a `atop`. U těchto nástrojů ovšem nastal problém s interpretací naměřených hodnot. Programy sledují tři různé údaje, z nich ovšem žádný neuvádí přesně to, co jsem potřeboval zjistit.

7.3 Příklad 1

První příklad je tvořen průchodem tetraedrální sítí a řešením jednoduché úlohy. Zadáním této úlohy je najít takový tetrahedron v síti, jehož střed leží nejblíž těžišti celé triangulace (které se určí jako průměr středů všech tetraedrů po souřadnicích vážený objemy tetraedrů).

Algoritmus: *Příklad 1*

Vstup: Tetraedrální síť ze souboru.
Výstup: Objem tetraedru.

1. Načíst zdrojovou tetraedrální síť ze souboru.
2. Nechť $W = (0, 0, 0)$ a $w = 0$.
3. Pro každý tetrahedron v síti:
   (a) Spočítat souřadnice těžiště a objem.
   (b) K W přičíst souřadnice těžiště.
   (c) K w přičíst objem tetraedru.
4. Nechť $W = \frac{1}{w} \cdot W$ po složkách. (Tím se získá těžiště sítě.)
5. Po každý tetrahedron v síti:
   (a) Spočítat souřadnice těžiště a objem.
   (b) Vyčíslit vzdálenost d těžiště tetraedru od zjištěného těžiště sítě.
   (c) Pokud je vzdálenost d nižší než nejmenší dříve vyčíslená vzdálenost, uložit ji jako d_{min} a nastavit V na objem daného tetraedru.
6. Vrátit V jako výsledek.

V příkladu 1 byl použit volně dostupný dataset *Super Phoenix*[1]. Dataset specifikuje tetraedrální síť představující triangulaci nukleárního reaktoru s 2 896 vertexy a 12 936 tetraedry.

[1] http://www.cs.umd.edu/class/fall2010/cmsc741/datasets.html

Rozhraní knihovny CGAL však neumožnilo načíst síť v této podobě, a proto verze příkladu s použitím CGAL načítá z datasetu pouze vertexy a provádí vlastní triangulaci, vedoucí na síť s 18 132 tetraedry (v terminologii CGAL označovaných jako buňky čili cells). Měření časové složitosti u prvního příkladu zahrnuje pouze část kódu s cykly a výpočty, triangulace ani načítání ze souboru není v naměřené době běhu zahrnuto.

7.4 Příklady 2 a 3

Druhý a třetí příklad představují procedurální generování tetraedrální sítě.

Algoritmus: *Příklady 2 a 3*

Vstup: Parametry generování (souřadnice počátku, vektor kroku).
Výstup: Vygenerovaná triangulace krychle podle zadaných parametrů.

1. Inicializovat generátor sítě.
2. Smazat stávající síť (pokud je neprázdná).
3. Vygenerovat pravidelnou mřížku vertexů.
4. Vygenerovat síť tetraedrů s použitím vygenerovaných vertexů.

Třetí příklad se liší od druhého jen velikostí generované sítě. V příkladu 2 se krychle rozdělí na $12 \times 12 \times 12$ (tedy 1 728) subkrychlí, v příkladu 3 jen na $10 \times 10 \times 10$ (tedy 1 000) subkrychlí. Každá subkrychle ja pak reprezentována šesti tetraedry.

7.5 Výsledky experimentů a vyhodnocení

Výsledky měření časové složitosti jsou uvedeny v tabulce 7.1, odpovídající grafy jsou uvedeny jako obrázky 7.1, 7.2 a 7.3. Výsledky měření prostorové složitosti jsou uvedeny v tabulce 7.2 a jako obrázky 7.4, 7.5 a 7.6.

V této sekci budou diskutovány výsledky měření a uvedeny jejich pravděpodobné příčiny.

U každého příkladu jsem provedl pět samostatných měření, abych tím redukoval náhodné chyby. V tabulkách je uveden průměr z měření, rozptyl (čili střední kvadratická odchylka) a medián, který je pro srovnávání hodnot zvlášť vhodný.

7.5.1 Časová složitost

Měření ukázalo, že OTK poskytuje rychlejší průchod sítí než CGAL – v prvním příkladu dosáhla lepších výsledků, a to dokonce i po přepočtu na jeden tetrahedron (CGAL-verze pracuje s větším množstvím tetraedrů z důvodů popsaných v sekci 7.3). Vzhledem k neznalosti podrobností vnitřní implementace CGAL nemohu s jistotou určit, co přesně bylo příčinou toho, že průchod sítí v ní je pomalejší.

Naopak generování sítě je podle výsledků měření podstatně pomalejší v knihovně OTK. Domnívám se, že příčinou je rozdílný postup knihoven. CGAL trianguluje síť postupně, což je pravděpodobně příčinou silně kolísajícího využití paměti, zatímco OTK jednorázově alokuje pomocnou datovou strukturu, kterou pak opakovaně řadí, aby z ní vyextrahovala potřebné údaje.

Příklad	Průměr	Rozptyl	Medián
příklad 1 : CGAL	1,66690	$4,225 \cdot 10^{-4}$	1,67475
příklad 1 : OTK	1,17233	$1,178 \cdot 10^{-4}$	1,16978
příklad 2 : CGAL	0,947560	$0,9664 \cdot 10^{-4}$	0,944800
příklad 2 : OTK	6,26079	$15,65 \cdot 10^{-4}$	6,26851
příklad 3 : CGAL	0,574300	$6,502 \cdot 10^{-4}$	0,554350
příklad 3 : OTK	3,47015	$14,17 \cdot 10^{-4}$	3,48339

Tabulka 7.1: Srovnání časové náročnosti implementací (údaje jsou v sekundách).

Příklad	Průměr	Rozptyl	Medián
příklad 1 : CGAL	1 066 598 B	$101 \cdot 10^9$	905 216 B
příklad 1 : OTK	777 421 B	$20 \cdot 10^9$	745 472 B
příklad 2 : CGAL	570 982 B	$5,3 \cdot 10^9$	618 496 B
příklad 2 : OTK	222 003 B	$11 \cdot 10^9$	237 568 B
příklad 3 : CGAL	294 912 B	$5,13 \cdot 10^9$	253 952 B
příklad 3 : OTK	120 422 B	$0,08053 \cdot 10^9$	126 976 B

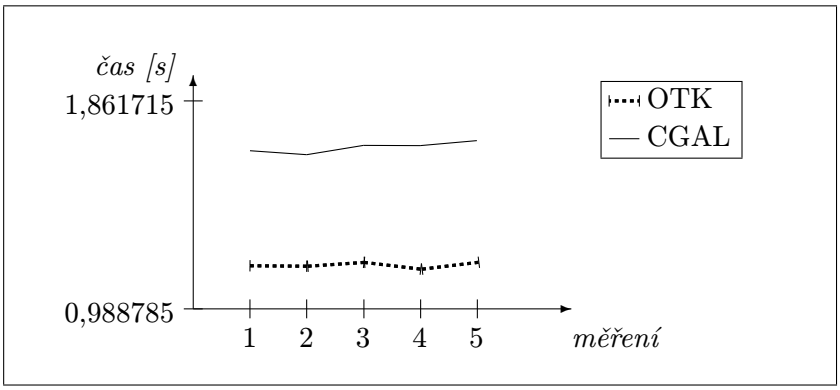
Tabulka 7.2: Srovnání paměťové náročnosti implementací (údaje jsou v bajtech).

Měření časové složitosti bylo vzhledem k použité metodě pracné, ale proběhlo bez větších problémů a vzhledem k relativně k nízkému rozptylu mají naměřené hodnoty dostatečnou vypovídací hodnotu.

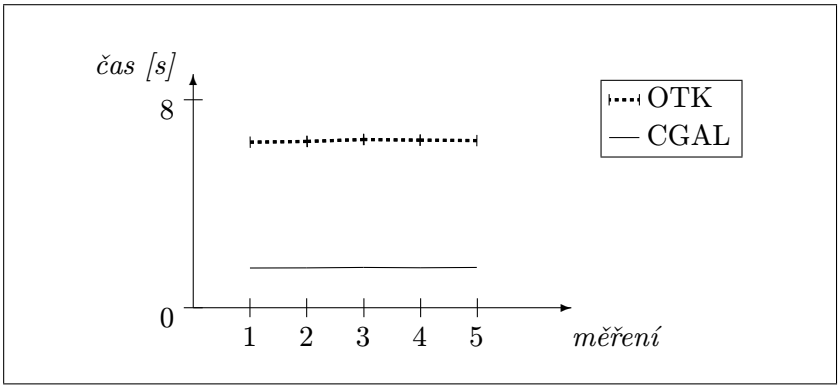
7.5.2 Prostorová složitost

Podle rozptylu hodnot při jednotlivých měřeních prostorové složitosti docházelo k velmi značným výkyvům spotřeby paměti u obou knihoven. Tyto výkyvy jsou nejvíce patrné na obrázcích 7.4 a 7.5. U knihovny CGAL docházelo k větším výkyvům než u OTK, což bylo pravděpodobně způsobeno vnitřní implementací knihovny. Vzhledem k tomu, že jsem nestudoval podrobnosti vnitřní implementace CGAL (seznámil jsem se pouze s jejím rozhraním a klíčovými vlastnostmi), přesná příčina tohoto kolísání mi není známa; domnívám se však, že hlavní příčinou byly časté alokace pomocné paměti. Další příčinou může být nespolehlivost zvolené metody měření spotřeby paměti.

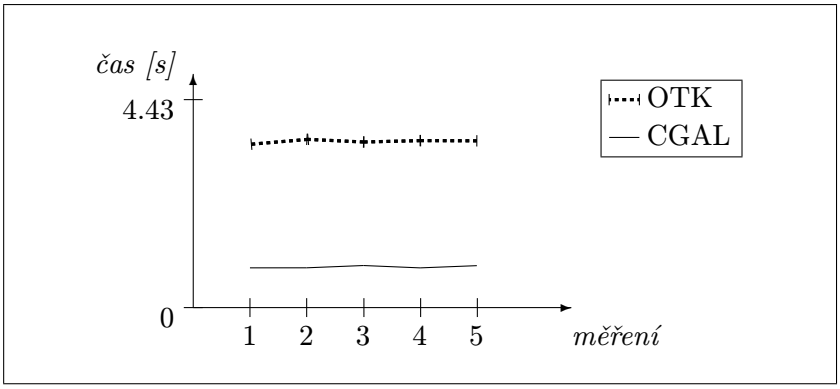
Naměřené hodnoty nelze interpretovat přímo jako paměť spotřebovanou knihovnou, protože zahrnují i paměť operačního systému potřebnou k vlastnímu běhu procesů. Přesto lze konstatovat, že OTK spotřebovala ve všech měřených případech méně paměti než CGAL; rozdíl je okolo 25 bajtů na tetrahedron.



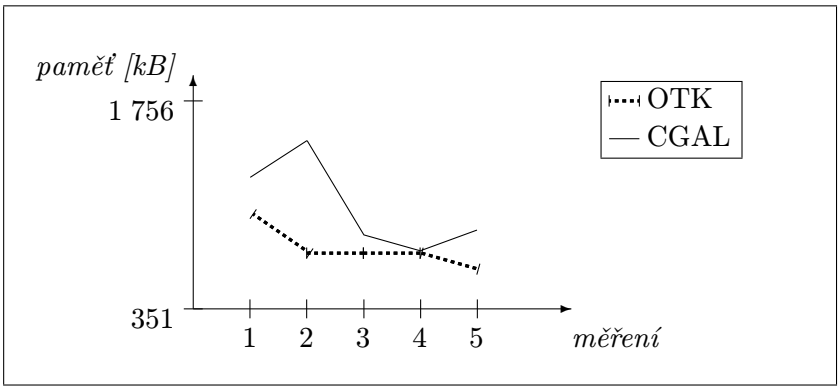
Obrázek 7.1: Výsledky měření časové složitosti 1. příkladu



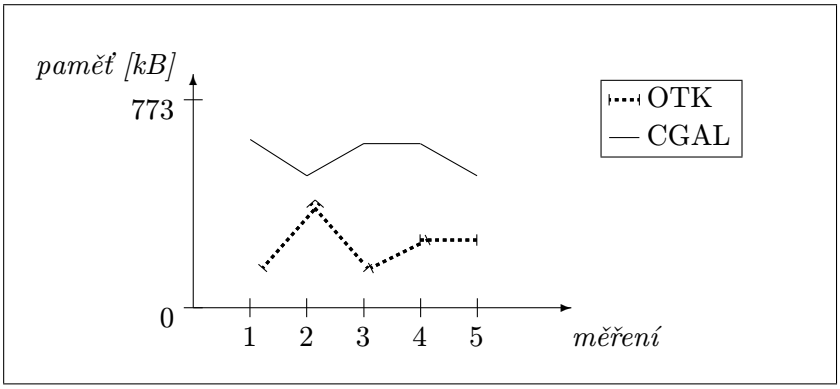
Obrázek 7.2: Výsledky měření časové složitosti 2. příkladu



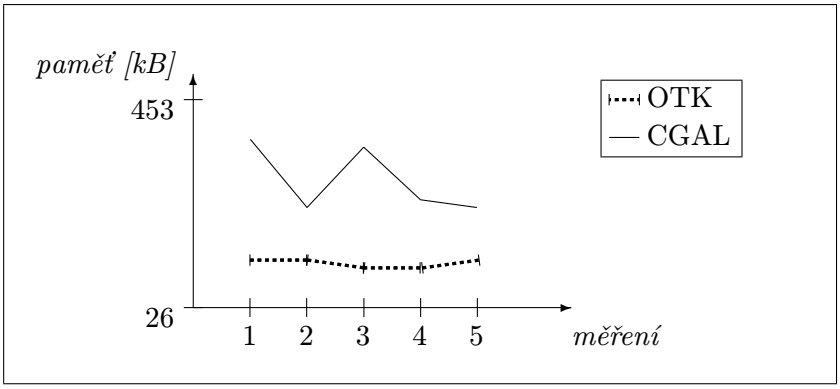
Obrázek 7.3: Výsledky měření časové složitosti 3. příkladu



Obrázek 7.4: Výsledky měření prostorové složitosti 1. příkladu



Obrázek 7.5: Výsledky měření prostorové složitosti 2. příkladu



Obrázek 7.6: Výsledky měření prostorové složitosti 3. příkladu

Kapitola 8

Možná rozšíření a aplikace

Knihovna OTK, navržená a implementovaná v rámci této diplomové práce, poskytuje sice velké množství funkcí, ale stále ne všechny, které by pro určité způsoby zpracování tetraedrální sítě mohly být potřeba. V této kapitole nejprve shrnu aplikace, pro které je knihovna vhodná již ve své stávající podobě. Dále uvedu příklady vhodných rozšíření, tedy funkcí, které by, podle mého názoru, bylo vhodné při dalším vývoji knihovny doimplementovat, aby se tak zvýšila praktická použitelnost knihovny a rozšířil seznam aplikací, pro něž je knihovna vhodná; případně oblastí vhodných k optimalizaci.

8.1 Analýza sítě metodami procházení stavového prostoru

Knihovna je vhodná pro metody BFS, DFS, backtracking apod. Metody BFS a DFS budou těžit ze snadného nalezení sousedních tetraedrů a možnosti uchovávat odkazy na jednotlivé elementy ve formě handles. Backtracking namísto toho může využít výjimečnou schopnost knihovny OTK uchovat až šest uživatelských bitů na tetrahedron bez potřeby další paměti. (Knihovna pro tento účel využívá část datové struktury, která by jinak zůstala nevyužitá.)

8.2 Výtvarné počítačové umění

Knihovna OTK nepracuje s geometrií sítě, díky tomu je geometricky flexibilní, a to v tom smyslu, že uživatel se může snadno rozhodnout pro použití více- či méně-dimenzionálních vektorových prostorů, případně pro použití hyperbolické či parabolické geometrie. Navíc hranice tetraedrů mohou být reprezentovány parametrickými plochami či mohou být dynamicky dopočítávány v závislosti na vývoji parametrů v okolí příslušného tetraedru. Tato flexibilita poskytuje široký prostor pro experimenty s cílem dosáhnout vyšší estetické kvality obrazového výstupu.

Snadné přidávání tetraedrů na okrajích sítě umožňuje vizualizovat růst. Systém dynamických atributů umožňuje snadné barvení či texturování sítě ve vrcholech i buňkách. Výsledný obraz či video může vzniknout např. řezem sítí.

8.3 „Celulární“ simulace

Celulární simulací rozumím takovou simulaci, při níž se nemění topologie buněk a nový stav každé buňky závisí jen na ní a jejím lokálním okolí. Knihovna OTK je vhodná pro tento

druh simulací, protože poskytuje prostředky pro efektivní průchod sítí včetně průchodu tetraedrů v lokálním okolí.

8.4 Procedurální generování tetraedrální sítě

Možnosti generování sítě jsou knihovnou OTK podporovány omezeně. Knihovna je vhodná zejména pro inkrementální metody, protože poskytuje rozhraní pro postupné přidávání jednotlivých tetraedrů na okraj sítě.

8.5 Navrhovaná rozšíření

V této sekci je uvedeno několik návrhů pro budoucí vývoj knihovny, jejichž realizace by ji učinila lépe použitelnou.

8.5.1 Dotazy one_to_any

Knihovna OTK poskytuje funkci `one_to_all<>()`, která umožňuje vytvoření množiny elementů zvoleného typu příslušných určitým způsobem ke konkrétnímu elementu téhož či jiného typu (např. všechny tetraedry incidentní s určitým vertexem, všechny halffaces incidentní se zadanou hranou apod.). Někdy však navržený algoritmus nepotřebuje znát všechny takové prvky, ale stačí jeden, případně stačí jen informace, zda takový prvek existuje. Tyto případy lze zatím řešit použitím funkce `one_to_all<>()` s následným testem množiny na prázdnost (`is_empty()`), ale takové řešení je neefektivní.

8.5.2 Dotazy incidence

Dalším vhodným rozšířením by bylo přidání podpory dotazů, zda jsou dva elementy sítě incidentní (např. tetrahedron a vertex). Tyto případy je ve stávající verzi řešit pomocí množin a funkce `one_to_all<>()`, ale toto řešení je neefektivní. Výkon knihovny v daných aplikacích by zvýšilo přidání samostatné optimalizované šablonové funkce `is_incident<>()`.

8.5.3 Lazy evaluation pro množiny a multimnožiny

Množiny a multimnožiny elementů tvoří podstatný subsystém knihovny OTK. Jejich současná implementace ovšem může trpět nízkou efektivitou, protože dochází k vytváření dočasných objektů a není možno odoptimalizovat jejich vytvoření ani alokaci související paměti. Tyto problémy je možno odstranit implementací zpožděného vyhodnocování (lazy evaluation).

Zásadní rozdíl by vznikl v případě, kdy jsou na určité množiny aplikovány množinové operátory a výsledek je pouze testován funkcí `elem()`, nebo `is_empty()`. V takovém případě by se nemusela vůbec alokovat paměť, ale program by mohl rovnou hledat první element vyhovující kritériu a vrátit ten bez potřeby dynamické alokace paměti.

8.5.4 Cirkulátory a další typy iterátorů

Stávající verze knihovny OTK poskytuje jaderné iterátory. Bylo by však vhodné poskytnout i přeskakující iterátory, které při každém posunu testují, zda se nedostaly na element označený ke smazání, a takový element případně přeskočí. Přítomnost přeskakujících iterátorů

je třeba dočasně řešit přidáním testu na platnost elementu (nikoliv na platnost handle!) do uživatelského kódu.

Cirkulátory jsou zvláštní iterátory, které se za posledním prvkem vrací zpět na první. Neexistuje tedy cirkulátor ukazující za konec kontejneru, ani žádný cirkulátor po prázdné množině.

8.5.5 Optimalizace algoritmů

Knihovna OTK v traitech umožňuje uživateli zvolit, že jeho sítě budou vždy manifold. Některé algoritmy je možno pro manifold sítě implementovat efektivněji než pro obecné sítě. Stávající verze knihovny však, pokud vím, této možnosti nevyužívá.

Kapitola 9

Závěr

V rámci této diplomové práce jsem v jazyce C++ vyvinul knihovnu OTK, umožňující práci s obecnými tetraedrálními sítěmi. Podle naměřených výsledků přitom knihovna OTK spotřebuje méně paměti než v současnosti používaná knihovna CGAL, což ji činí vhodnou pro aplikační oblasti vyžadující zpracování velkých sítí nebo pracujících s výrazně limitovaným množstvím dostupné paměti.

OTK je univerzálnější než CGAL, protože umožňuje práci s obecnými tetraedrálními sítěmi (CGAL poskytuje nástroje zejména pro generování a zpracování triangulací). Na druhou stranu je OTK ve většině případů několikanásobně pomalejší než CGAL, zejména se to ukázalo v případě generování sítě.

Knihovna OTK je vhodná zejména pro analýzu sítě průchodem (např. analýzu medicínských dat), výtvarné počítačové umění, „celulární“ simulace (např. metodu konečných prvků) a procedurální generování tetraedrální sítě nějakou inkrementální metodou. V rámci dalšího vývoje by knihovna měla být rozšířena o podporu dotazů typu *one-to-any* a dotazů incidence. Všechny algoritmy by měly být podle možností optimalizovány pro zvýšení výkonu.

Zvláštní přínos spatřuji v podpoře množin elementů umožňující pohodlnou (a do budoucna snad i efektivní) práci se sítěmi s využitím množinových operací.

Při vývoji knihovny jsem použil vodopádový model vývoje softwaru. Použitý model se neosvědčil, protože jsem přecenil komplikovanost pozdějších fází vývoje. V případě použití inkrementálního modelu by výsledkem práce pravděpodobně byla knihovna s nižším množstvím funkcí, ale přehlednější a lépe zdokumentovaná.

Výsledky měření ukazují, že OTK má ve všech případech nižší paměťovou složitost než CGAL. (Mluví se zde o absolutní složitosti vzorových příkladů, nikoliv o asymptotické složitosti použitých algoritmů.) V časové složitosti vykazoval CGAL lepší výsledky, takže algoritmy používané knihovnou OTK by měly být optimalizovány k dosažení lepší časové složitosti, bude-li to možné. Knihovna OTK ve srovnání s CGAL umožňuje zpracování obecných tetraedrálních sítí (CGAL se zaměřuje především na triangulace), což ji předurčuje pro účely, kdy je potřeba zpracovávat netypické tetraedrální sítě.

Literatura

[1] Tetrahedron [online]. http://en.wikipedia.org/wiki/Tetrahedron, 2013-03-25 [cit. 2013-03-26].

[2] Delaunay triangulation [online]. http://en.wikipedia.org/wiki/Delaunay_triangulation, 2013-04-21 [cit. 2013-04-28].

[3] free(1) - Linux man page [online]. http://linux.die.net/man/1/free, [cit. 2013-04-05].

[4] Alumbaugh, T. J.; Jiao, X.: Compact Array-Based Mesh Data Structures. In *Proceedings, 14th International Meshing Roundtable, Springer-Verlag*, Center for Simulation of Advanced Rockets, 2005, s. 485–503.

[5] George, P.-L.; Borouchaki, H.: *Delaunay Triangulation and Meshing*. HERMES, 1998, ISBN 2-86601-692-0.

[6] Giezeman, G.-J.; Veltkamp, R.; Wesselink, W.: Getting Started with CGAL 2.1 [online]. http://graphics.stanford.edu/courses/cs368-00-spring/manuals/CGAL_Tutorial.pdf, 1999-12-01.

[7] Josuttis, N. M.: *C++ Standardní knihovna a STL: Kompletní průvodce*. CP Books, 2005, ISBN 80-251-0511-1.

[8] McConnell, S.: *Dokonalý kód*. Computer Press, a.s., 2005, ISBN 80-251-0849-X.

[9] Pion, S.; Teillaud, M.: CGAL Manual: Chapter 39. http://www.cgal.org/Manual/latest/doc_html/cgal_manual/Triangulation_3/Chapter_main.html, 2012-10-23 [cit. 2013-03-26].

[10] Šiler, O.: Vector Entity: Getting Started [online]. http://mdstk.sourceforge.net/documentation.html → Vector Entity: Getting Started, 2007-11-01.

[11] Žára, J.; Beneš, B.; Sochor, J.; aj.: *Moderní počítačová grafika*. Computer Press, a.s., 2010, ISBN 80-251-0454-0.

Příloha A

Plakat

OTK

David Hromádka
<xhroma06@stud.fit.vutbr.cz>

KNIHOVNA PRO PRÁCI S TETRAEDRÁLNÍ SÍTÍ
DIPLOMOVÁ PRÁCE (2013)

CÍL:

- Vyvinout paměťově úspornou knihovnu pro práci s tetraedrální sítí

VÝSLEDKY:

- Knihovna OTK spotřebuje na stejnou síť méně paměti než CGAL

APLIKAČNÍ OBLASTI:

- Analýza prostorových dat metodami procházení stavového prostoru
- Metoda konečných prvků (FEM)
- Procedurální generování tetraedrální sítě